Università di Genova

Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi

An Unexpected Journey: Towards Runtime Verification of Multiagent Systems and Beyond

Angelo Ferrando

Theses Series DIBRIS-TH-2019-2

Università di **Genova**

Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

Ph.D. Thesis in
Computer Science and Systems Engineering
Computer Science Curriculum

An Unexpected Journey: Towards Runtime Verification of Multiagent Systems and Beyond

by

Angelo Ferrando

February, 2019

Ph.D. Thesis in Computer Science and Systems Engineering (S.S.D. INF/01)
Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi
Università di Genova

Candidate
Angelo Ferrando
`angelo.ferrando@dibris.unige.it`

Title
An Unexpected Journey:
Towards Runtime Verification of Multiagent Systems and Beyond

Advisors
Davide Ancona
DIBRIS, Università di Genova
`davide.ancona@unige.it`

Viviana Mascardi
DIBRIS, Università di Genova
`viviana.mascardi@unige.it`

External Reviewers
Rafael H. Bordini
Pontificia Universidade Católica do Rio Grande do Sul
`r.bordini@pucrs.br`

Frank De Boer
Universiteit Leiden
`f.s.de.boer@liacs.leidenuniv.nl`

Louise A. Dennis
University of Liverpool
`l.a.dennis@liverpool.ac.uk`

Location
DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy

Submitted On
February 2019

Dedicated to my family
...all right, all right, also to Martina!

“I’m going on an adventure!”
- J.R.R. Tolkien, The Hobbit

Abstract

The Trace Expression formalism derives from works started in 2012 and is mainly used to specify and verify interaction protocols at runtime, but other applications have been devised. More specifically, this thesis describes how to extend and apply such formalism in the engineering process of distributed artificial intelligence systems (such as Multiagent systems).
This thesis extends the state of the art through four different contributions:

1. *Theoretical*: the thesis extends the original formalism in order to represent also parametric and probabilistic specifications (*parametric trace expressions* and *probabilistic trace expressions* respectively).

2. *Algorithmic*: the thesis proposes algorithms for verifying trace expressions at runtime in a decentralized way. The algorithms have been designed to be as general as possible, but their implementation and experimentation address scenarios where the modelled and observed events are communicative events (interactions) inside a multiagent system.

3. *Application*: the thesis analyzes the relations between runtime and static verification (e.g. model checking) proposing hybrid integrations in both directions. First of all, the thesis proposes a trace expression model checking approach where it shows how to statically verify LTL property on a trace expression specification. After that, the thesis presents a novel approach for supporting static verification through the addition of monitors at runtime (post-process).

4. *Implementation*: the thesis presents RIVERtools, a tool supporting the writing, the syntactic analysis and the decentralization of trace expressions.

Publications

Articles

Ancona, Davide, Angelo Ferrando, and Viviana Mascardi (2018b). “Improving flexibility and dependability of remote patient monitoring with agent-oriented approaches”. In: *Int. J. Agent-Oriented Software Engineering* 6, Nos. 3/4.

Ancona, Davide, Daniela Briola, Angelo Ferrando, and Viviana Mascardi (2015b). “Runtime verification of fail-uncontrolled and ambient intelligence systems: A uniform approach”. In: *Intelligenza Artificiale* 9.2, pp. 131–148. DOI: 10.3233/IA-150084. URL: https://doi.org/10.3233/IA-150084.

Del Fatto, Vincenzo, Gabriella Dodero, Armin Bernhard, Angelo Ferrando, Davide Ancona, Viviana Mascardi, Robert Laurini, and Giuseppe Roccasalva (2017). “Hackmytown: an Educational Experience on Smart Cities”. In: *IxD&A* 32, pp. 153–164. URL: http://www.mifav.uniroma2.it/inevent/events/idea2010/index.php?s=10\&\#38;a=10\&\#38;link=ToC_32_P\&\#38;link=32_9_abstract.

Ferrando, Angelo (2019). “The early bird catches the worm: First verify, then monitor!” In: *Science of Computer Programming* 172, pp. 160 –179. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2018.11.008. URL: http://www.sciencedirect.com/science/article/pii/S0167642318304349.

Conference Proceedings

Aielli, Federica, Davide Ancona, Pasquale Caianiello, Stefania Costantini, Giovanni De Gasperis, Antinisca Di Marco, Angelo Ferrando, and Viviana Mascardi (2016). “FRIENDLY & KIND with your Health: Human-Friendly Knowledge-INtensive Dynamic Systems for the e-Health Domain”. In: *Highlights of Practical Applications of Scalable Multi-Agent Systems. The PAAMS Collection - International Workshops of PAAMS 2016, Sevilla, Spain, June 1-3, 2016. Proceedings.* Ed. by Javier Bajo, María José Escalona, Sylvain Giroux, Patrycja Hoffa-Dabrowska, Vicente Julián, Paulo Novais, Nayat Sánchez Pi, Rainer Unland, and Ricardo Azambuja Silveira. Vol. 616. Communications in Computer and Information Science. Springer, pp. 15–26. DOI: 10.1007/978-3-319-39387-2_2. URL: https://doi.org/10.1007/978-3-319-39387-2_2.

Ancona, Davide, Angelo Ferrando, and Viviana Mascardi (2016). “Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification”. In: *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday.* Ed. by Erika Ábrahám, Marcello M.

Bonsangue, and Einar Broch Johnsen. Vol. 9660. Lecture Notes in Computer Science. Springer, pp. 47–64. DOI: 10.1007/978-3-319-30734-3_6. URL: https://doi.org/10.1007/978-3-319-30734-3_6.

Ancona, Davide, Angelo Ferrando, and Viviana Mascardi (2017). "Parametric Runtime Verification of Multiagent Systems". In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*. Ed. by Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee. ACM, pp. 1457–1459. URL: http://dl.acm.org/citation.cfm?id=3091328.

– (2018a). "Agents Interoperability via Conformance Modulo Mapping". In: *Proceedings of the 19th Workshop "From Objects to Agents", Palermo, Italy, June 28-29, 2018*. Ed. by Massimo Cossentino, Luca Sabatucci, and Valeria Seidita. Vol. 2215. CEUR Workshop Proceedings. CEUR-WS.org, pp. 109–115. URL: http://ceur-ws.org/Vol-2215/paper_18.pdf.

Ancona, Davide, Daniela Briola, Angelo Ferrando, and Viviana Mascardi (2015a). "Global Protocols as First Class Entities for Self-Adaptive Agents". In: *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, Istanbul, Turkey, May 4-8, 2015*. Ed. by Gerhard Weiss, Pinar Yolum, Rafael H. Bordini, and Edith Elkind. ACM, pp. 1019–1029. URL: http://dl.acm.org/citation.cfm?id=2773282.

– (2016). "MAS-DRiVe: a Practical Approach to Decentralized Runtime Verification of Agent Interaction Protocols". In: *Proceedings of the 17th Workshop "From Objects to Agents" co-located with 18th European Agent Systems Summer School (EASSS 2016), Catania, Italy, July 29-30, 2016*. Ed. by Corrado Santoro, Fabrizio Messina, and Massimiliano De Benedetti. Vol. 1664. CEUR Workshop Proceedings. CEUR-WS.org, pp. 35–43. URL: http://ceur-ws.org/Vol-1664/w7.pdf.

Ancona, Davide, Angelo Ferrando, Luca Franceschini, and Viviana Mascardi (2017). "Parametric Trace Expressions for Runtime Verification of Java-Like Programs". In: *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, Barcelona , Spain, June 20, 2017*. ACM, 10:1–10:6. DOI: 10.1145/3103111.3104037. URL: http://doi.acm.org/10.1145/3103111.3104037.

– (2018a). "Coping with Bad Agent Interaction Protocols When Monitoring Partially Observable Multiagent Systems". In: *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection - 16th International Conference, PAAMS 2018, Toledo, Spain, June 20-22, 2018, Proceedings*. Ed. by Yves Demazeau, Bo An, Javier Bajo, and Antonio Fernández-Caballero. Vol. 10978. Lecture Notes in Computer Science. Springer, pp. 59–71. DOI: 10.1007/978-3-319-94580-4_5. URL: https://doi.org/10.1007/978-3-319-94580-4_5.

– (2018b). "Managing Bad AIPs with RIVERtools". In: *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection - 16th International Conference, PAAMS 2018, Toledo, Spain, June 20-22, 2018, Proceedings*. Ed. by Yves Demazeau, Bo An, Javier Bajo, and

Antonio Fernández-Caballero. Vol. 10978. Lecture Notes in Computer Science. Springer, pp. 296–300. DOI: 10.1007/978-3-319-94580-4_24. URL: https://doi.org/10.1007/978-3-319-94580-4_24.

Beux, Silvio et al. (2015). "Computational thinking for beginners: A successful experience using Prolog". In: *Proceedings of the 30th Italian Conference on Computational Logic, Genova, Italy, July 1-3, 2015.* Ed. by Davide Ancona, Marco Maratea, and Viviana Mascardi. Vol. 1459. CEUR Workshop Proceedings. CEUR-WS.org, pp. 31–45. URL: http://ceur-ws.org/Vol-1459/paper10.pdf.

Ferrando, Angelo (2015). "Parametric protocol-driven agents and their integration in JADE". In: *Proceedings of the 30th Italian Conference on Computational Logic, Genova, Italy, July 1-3, 2015.* Ed. by Davide Ancona, Marco Maratea, and Viviana Mascardi. Vol. 1459. CEUR Workshop Proceedings. CEUR-WS.org, pp. 72–84. URL: http://ceur-ws.org/Vol-1459/paper26.pdf.

– (2016). "Automatic Partitions Extraction to Distribute the Runtime Verification of a Global Specification". In: *Proceedings of the Doctoral Consortium of AI*IA 2016 co-located with the 15th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2016), Genova, Italy, November 29, 2016.* Ed. by Viviana Mascardi and Ilaria Torre. Vol. 1769. CEUR Workshop Proceedings. CEUR-WS.org, pp. 40–45. URL: http://ceur-ws.org/Vol-1769/paper07.pdf.

Ferrando, Angelo, Davide Ancona, and Viviana Mascardi (2016). "Monitoring Patients with Hypoglycemia Using Self-adaptive Protocol-Driven Agents: A Case Study". In: *Engineering Multi-Agent Systems - 4th International Workshop, EMAS 2016, Singapore, Singapore, May 9-10, 2016, Revised, Selected, and Invited Papers.* Ed. by Matteo Baldoni, Jörg P. Müller, Ingrid Nunes, and Rym Zalila-Wenkstern. Vol. 10093. Lecture Notes in Computer Science. Springer, pp. 39–58. DOI: 10.1007/978-3-319-50983-9_3. URL: https://doi.org/10.1007/978-3-319-50983-9_3.

– (2017). "Decentralizing MAS Monitoring with DecAMon". In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017.* Ed. by Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee. ACM, pp. 239–248. URL: http://dl.acm.org/citation.cfm?id=3091164.

Ferrando, Angelo, Silvio Beux, Viviana Mascardi, and Paolo Rosso (2016). "Identification of Disease Symptoms in Multilingual Sentences: An Ontology-Driven Approach". In: *Proceedings of the First Workshop on Modeling, Learning and Mining for Cross/Multilinguality (MultiLingMine 2016) co-located with the 38th European Conference on Information Retrieval (ECIR 2016), Padova, Italy, March 20, 2016.* Ed. by Dino Ienco, Mathieu Roche, Salvatore Romeo, Paolo Rosso, and Andrea Tagarelli. Vol. 1589. CEUR Workshop Proceedings. CEUR-WS.org, pp. 6–15. URL: http://ceur-ws.org/Vol-1589/MultiLingMine1.pdf.

Ferrando, Angelo, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi (2018a). "Recognising Assumption Violations in Autonomous

Systems Verification". In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*. Ed. by Elisabeth André, Sven Koenig, Mehdi Dastani, and Gita Sukthankar. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, pp. 1933–1935. URL: http://dl.acm.org/citation.cfm?id=3238028.

Ferrando, Angelo, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi (2018b). "Verifying and Validating Autonomous Systems: Towards an Integrated Approach". In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*. Ed. by Christian Colombo and Martin Leucker. Vol. 11237. Lecture Notes in Computer Science. Springer, pp. 263–281. DOI: 10.1007/978-3-030-03769-7_15. URL: https://doi.org/10.1007/978-3-030-03769-7_15.

Acknowledgements

I want to thank prof. Michael Fisher and prof. Louise A. Dennis for letting me visit their laboratory at the University of Liverpool (UK).
I want to thank prof. Amit K. Chopra for letting me visit his laboratory at the University of Lancaster (UK).
I also want to thank prof. Michael Winikoff and prof. Stephen Cranefield for letting me visit their laboratory at the University of Otago (New Zealand).

Now for all the others, in Italian!

In questo mattone di inglese, almeno i ringraziamenti lasciamoli in italiano no? Vediamo un po' chi purtroppo si è meritato di essere ringraziato...

Ringrazio i miei relatori Viviana e Davide per essere stati presenti in questi 4 anni (laurea magistrale e Ph.D.) e per avermi sempre lasciato libertà di scelta sugli argomenti su cui fare ricerca.
Ringrazio la mia famiglia per avermi supportato sia nel periodo della laurea triennale e magistrale, sia durante il dottorato. Una famiglia normale avrebbe cercato di non farmi iniziare un nuovo percorso universitario dopo la laurea magistrale, grazie per non esserlo stata.
Ringrazio Martina per avermi sopportato[1] per tutto questo tempo, ci siamo conosciuti in università quando frequentavo il primo anno di Informatica a Genova e da allora siamo sempre stati insieme. Non avrei mai potuto ottenere gli stessi risultati senza di lei e, come dissi nei ringraziamenti della tesi magistrale, lei resterà sempre la mia vittoria più grande. Un dottorato è niente a confronto!
Ringrazio tutti i miei amici più cari: Paolo (aka *Monta*), Davide (aka *Pappy*), Andrea (aka *Brogne*), Michele (aka *Miky*), Ilaria (aka *Ila bionda*), Ilaria (aka *Pingu*), Eleonora e Alice. Gli anni sono passati ma voi no[2] e sono grato di ogni momento passato insieme.
Ringrazio tutti gli altri dottorandi dell'università di Genova, in particolare Federico, Laura e Tommaso. Finalmente siamo arrivati alla fine di questa lunga maratona iniziata insieme al primo anno e sono onorato di aver intrapreso questo cammino con voi.
Ringrazio Lorenzo (aka prof. Repetto) per essere sempre stato la mia fonte di ispirazione. Se non fosse stato per lui non avrei nemmeno iniziato l'università.

[1]Con la "o" esatto! Non è un errore di battitura.
[2]Solo invecchiati.. male per di più.. io invece sono invecchiato benissimo!

Contents

I Introduction 1

1 Motivations and Aims of the Thesis 2

2 Structure and Contributions of the Thesis 6
2.1 Structure . . . 6
2.2 Contributions . . . 7
2.2.1 Propose an expressive formalism and its extensions to support runtime verification of complex systems . . . 8
2.2.2 Design and define the algorithms for obtaining a decentralized runtime verification approach of Agent Interaction Protocols . . . 8
2.2.3 Propose a hybrid approach combining runtime and static verification of multiagent systems . . . 10
2.2.4 Present a tool developed for supporting runtime verification guided by trace expressions and a case study . 10
2.3 How to read the Thesis . . . 11

II Background 14

3 Preliminaries 15
3.1 Multiagent Systems and Distributed Artificial Intelligence . . 16
3.2 Rational agents . . . 20
3.3 Jason . . . 20
3.4 JADE . . . 21
3.4.1 Architecture . . . 21
3.4.2 Creating agents . . . 22
3.5 Techniques for checking the system's behaviour . . . 27
3.5.1 Model checking . . . 28
3.5.2 Runtime verification . . . 29
3.5.3 Runtime verification versus Model checking . . . 30
3.6 LTL . . . 31
3.6.1 LTL syntax and semantics . . . 32
3.6.2 Non deterministic Büchi Automaton . . . 32
3.6.3 LTL Model Checking . . . 33
3.6.4 Automata-Based Model Checking . . . 33
3.6.5 LTL_3 . . . 33
3.7 Model Checking Agent Programming Languages . . . 35
3.8 Hidden Markov Models . . . 36

3.9 Global Types 37
3.9.1 Syntax 38
3.9.2 Semantics 38

4 Trace expressions **40**
4.1 Introduction 41
4.2 The trace expression formalism 42
4.2.1 Events 42
4.2.2 Event types 42
4.2.3 Trace expressions 43
4.2.4 Deterministic trace expressions 47
4.2.5 Expansive trace expressions 48
4.2.6 Derived operators 48
4.3 Examples of specifications with trace expressions 49
4.3.1 Ping Pong Protocol 49
4.3.2 Alternating Bit Protocol 51
4.3.3 Non context free languages 52
4.4 Trace expressions monitoring 53
4.5 Comparison with LTL 55
4.5.1 Comparing trace expressions and LTL 55

5 State of the art **58**
5.1 Engineering Multiagent Systems 59
5.2 Blindingly Simple Protocol Language 61
5.3 Commitment-based Interaction 62
5.4 New Hierarchical Agent Protocol Notation 64
5.5 Self-adaptive Systems Engineering 64

III Formalism extensions **67**

6 Parametric Trace Expressions **68**
6.1 Introduction 69
6.1.1 Illustrative example 69
6.2 Formalization 70
6.3 Illustrative example revisited 72
6.4 Case study 74
6.4.1 Informal specification of the protocol 74
6.4.2 Formal specification of the protocol 74
6.5 Discussion 78

7 Probabilistic Trace Expressions **79**
7.1 Introduction 80
7.2 Runtime Verification with State Estimation 82
7.3 Probabilistic Trace Expressions 82
7.3.1 Non Determinism in State Transitions 86

7.3.2 From Trace Expressions to Probabilistic Trace Expressions . . . 88
7.4 From Hidden Markov Models to Probabilistic Trace Expressions 89
7.4.1 The HMM2PTE Algorithm . . . 89
7.4.2 Forward Algorithm for Probabilistic Trace Expressions 89
7.4.3 Satisfying LTL Properties when Gaps Are Observed . 93
7.5 Implementation and Experiments . . . 94
7.6 Discussion . . . 95

IV Engineering Agent Interaction Protocols 96

8 Issues with Agent Interaction Protocols 97
8.1 Introduction . . . 98
8.2 Projection of an Agent Interaction Protocol . . . 99
8.3 State of the art . . . 100
8.4 The Good, the Bad and the Ugly . . . 101
8.5 Partial Observability: how the Good Becomes Bad . . . 104
8.5.1 Observability-driven transformation of trace expressions 106
8.5.2 Implementation and Experiments . . . 107
8.6 Revisiting Good and Bad notions . . . 108
8.7 Discussion . . . 109

9 Decentralized Runtime Verification of Agent Interaction Protocols 110
9.1 Introduction . . . 111
9.2 Motivations . . . 111
9.3 DecAMon: a Gentle Introduction . . . 114
9.3.1 High-level Description and Examples . . . 115
9.4 Design . . . 118
9.5 Implementation and Experiments . . . 122
9.6 Discussion . . . 125

10 Decentralized Runtime Verification of Agent Interaction Protocols with Gaps 127
10.1 Introduction . . . 128
10.2 Exploiting DecAMon for PTEs . . . 129
10.3 Handling Gaps in Decentralized RV . . . 130
10.3.1 Synchronizing Decentralized Gaps Management . . . 132
10.4 Example . . . 133
10.5 Implementation and Experiments . . . 136
10.6 Discussion . . . 141

11 Conformance checking 142
11.1 Introduction . . . 143
11.2 State of the art . . . 146
11.3 LAIP Conformance Modulo Mapping . . . 147

11.4 Conformance algorithm: pseudo-code 151
11.5 Implementation and Experiments 154
11.6 Discussion 157

V Combining static and runtime verification 158

12 Trace expressions model checking 159
12.1 Introduction 160
12.2 State of the art 161
12.3 Motivations 163
12.4 Model Checking Trace expressions 164
12.4.1 1st step: Rewriting 165
12.4.2 2nd step: Translation 171
12.4.3 3rd step: Product 173
12.5 Implementation and Experiments 174
12.6 Discussion 177

13 Recognising Assumption Violations in Autonomous Systems Verification 178
13.1 Introduction 179
13.2 State of the art 180
13.3 Running Example 182
13.4 Recognising Assumption Violations 184
13.4.1 AJPF Static Formal Verification 185
13.4.2 Event Types for AJPF Environments 185
13.4.3 Abstract Model Generation 188
13.4.4 MCAPL Runtime Verification 190
13.5 Discussion 190

VI Implementation and Case Study 194

14 Development of a framework supporting trace expressions RV 195
14.1 Introduction 196
14.2 Trace expression RV using SWI-Prolog 197
14.3 RIVERtools 198
14.3.1 An example using RIVERtools 204
14.3.2 Decentralizing the Example with RIVERtools 207
14.3.3 Screenshots 212
14.4 Tutorial: How to use RIVERtools 214
14.4.1 How to install SWI-Prolog 214
14.4.2 How to install RIVERtools Eclipse plugin 215
14.4.3 How to use RIVERtools plugin (through an example) . 215
14.4.4 How to verify a MAS implemented in JADE 216
14.5 Discussion 217

15 Case Study **219**
15.1 Introduction 220
15.1.1 A Jason Framework Supporting Agents Driven by Parametric Trace Expressions 221
15.1.2 Modeling Clinical Guidelines 222
15.1.3 Modeling Management of Hypoglycemia in the Newborns 224
15.2 Experiments 228
15.2.1 A priori verification (attains fault tolerance and removal) 228
15.2.2 Self-adaptation (attains flexibility and fault tolerance and removal) 228
15.2.3 Performances 230
15.3 Discussion 232

VII Discussion **233**

16 Comparison **234**
16.1 Trace Expressions VS State of the art 234
16.1.1 Comparison 235

17 Conclusions and Future Work **237**
17.1 Conclusions 238
17.2 Expected Future Directions 239
17.3 Unexpected Future Directions 241

A Medical Guidelines **245**

List of Figures

2.1 Dependency graph of the main chapters. For instance, to understand Chapter 10 you have to read chapters 8 and 7 first. . . 13

3.1 Model checking procedure (Gluch et al., 2019) 28
3.2 FSM of the monitor for $p\,U q$, with $AP = \{p, q\}$ 34
3.3 Steps required to generate an FSM from an LTL formula φ . . 35
3.4 AJPF architecture (Bordini et al., 2008) 36

4.1 Operational semantics of trace expressions 45
4.2 Empty trace containment 45
4.3 An abstract view of how to build a monitor. 54

6.1 Transition system for parametric trace expressions 71

7.1 An example of HMM (from (Stoller et al., 2011)). 82
7.2 Transition system for probabilistic trace expressions states. . . 84
7.3 Rules for nondeterminism and transitive closure. 87

10.1 Centralized algorithm: changing number of agents. 138
10.2 Decentralized algorithm: changing number of agents. 138
10.3 Centralized algorithm: changing number of operators. 139
10.4 Decentralized algorithm: changing number of operators. . . . 139
10.5 Centralized algorithm: changing number of shuffled sub-PTEs. 140
10.6 Decentralized algorithm: changing number of shuffled sub-PTEs. 140

11.1 (a) and (b) MAS presented in the example; (c) *Buyer* substitutes *Client* through the interface i driven by $M_{\mathcal{A}} = \{Buyer \mapsto Client, Seller \mapsto BookShop\}$, $M_{\mathcal{M}} = \{res? \mapsto book?, res \mapsto book, money \mapsto ack, no \mapsto no_avbl\}$); (d) *Seller* substitutes *BookShop* with an interface driven by the same maps. 156
12.1 Büchi Automaton $B_{\tau'}$ 172

13.1 General view. 186
13.2 Trace expression template for generating abstract environments. 187
13.3 Trace Expressions for Constrs where $B_{j,i} \neq NB_{j,i}$. 188
13.4 Trace expression for a Cruise Control Agent. 190
13.5 Trace Expression for the Constraints on a Car where the driver only accelerates when it is safe to do so, and never uses both brake and acceleration pedal at the same time. 191

14.1 RIVERtools general representation (left) and its exploitation in three different scenarios: JADE, Jason and Node.js (right). . 199

14.2 RIVERtools exploited in JADE. 202
14.3 Abstract view of how JPL is used inside the JADE-Connector. 203
14.4 The book-shop scenario presented in Section 14.3.1. 212
14.5 Error: Partition not valid 213
14.6 Error: After having removed the role "frank", we have an existence error. 213
14.7 Error: After having removed the event type buy, we have an existence error. 214

15.1 Agents *patient1* and *patient2* sending the hours of life to agent *doctor1*. 229
15.2 Agent *doctor1* sending the threshold to *patient1*. 229
15.3 Agent *patient1* perceiving a low level of glucose. 230
15.4 Agent *doctor1* asking for a protocol switch to *patient1*. 230
15.5 Agent *patient1* displaying the need of an intravenous injection. 230
15.6 Agent *patient1* asking *doctor1* to intervene. 231

List of Tables

9.1 Experimental results: using DecAMon to extract minimal monitoring safe partitions. 124
9.2 Experimental results: filtering minimal monitoring safe partitions using different post-processing functions. 125

10.1 Average time of the centralized and decentralized algorithms; "sh. PTE" stands for "shuffled sub-PTE". 137

15.1 Experiments 231

16.1 Comparison 236

Listings

14.1 Book-shop trace expression written inside RIVERtools. 205
14.2 BookPurchase.java automatically generated by RIVERtools. . . 206
14.3 Manual decentralization of book-shop trace expression in RIVER-tools. 207
14.4 BookPurchase.java automatically generated by RIVERtools where we decentralize the RV on a fixed partition. 208
14.5 Automatic decentralization of book-shop trace expression in RIVERtools. 209
14.6 BookPurchase.java automatically generated by RIVERtools where we decentralize the RV on a not specified minimal monitoring safe partition. 211

Open source code

- https://github.com/AngeloFerrando/TExpSWIPrologConnector
  Java library for supporting the integration ("bridge") between trace expressions implemented using SWI-Prolog and Java. This library works as a connector for defining and querying trace expressions directly from Java, and it is used as the main pillar for building other system integrations.
- https://github.com/AngeloFerrando/TExpRVJade
  Java library for implementing Runtime Verification of multi-agent systems implemented in JADE using trace expressions. More specifically, trace expressions are used to define Agent Interaction Protocols that can be checked at runtime, both in a centralized and a decentralized way (based on the achievements of Chapter 9).
- https://github.com/AngeloFerrando/trace_expression_plugin_eclipse
  The IDE which has been developed for supporting the use of the trace expressions formalism in a more developer-friendly way (Tutorial available in Section 14.4).
- http://www.ParametricTraceExpr.altervista.org
  The SWI-Prolog and JADE code developed for using parametric trace expressions (this code is used inside the TExpSWIPrologConnector and TExpRVJade projects). The theory behind this implementation is presented in Chapter 6.
- http://trace2buchi.altervista.org
  The code implementing the translation from a trace expression to the corresponding Büchi Automaton. The theory behind this implementation is presented in Chapter 12.
- http://mcapl.sourceforge.net
  Full source code for the integration of trace expressions inside the MCAPL framework. The theory behind this implementation is presented in Chapter 13.

Acronyms

ACLs Agent Communication Languages

AIL Agent Infrastructure Layer

AIP Agent Interaction Protocol

AJPF Agent Java PathFinder

AOP Agent-Oriented Programming

API Application Programming Interface

BDI Beliefs Desires Intentions

BSPL Blindingly Simple Protocol Language

CM Commitment Machines

DRV Decentralized Runtime Verification

EASS Engineering Autonomous Space Software

HAPN Hierarchical Agent Protocol Notation

HMM Hiden Markov Model

IDE Integrated Development Environment

IoT Internet of Things

JPF Java PathFinder

LTL Linear Temporal Logic

MAS Multiagent System

MCAPL Model Checking Agent Programming Languages

RPM Remote Patient Monitoring

RV Runtime Verification

SOC Service-Oriented Computing

Part I

Introduction

This part is focused on presenting the contents of the thesis. We will first discuss the motivations of this work and we will then introduce the aims of the thesis and its structure. Finally, we will provide a section on how to read this work with a corresponding dependency graph for the various chapters.

1 *Motivations and Aims of the Thesis*

Nowadays, multiagent systems (MAS) are a very well known and established research field. Though MAS may seem young, huge advances have been made since their introduction in (Wooldridge, 1992; Wooldridge and Jennings, 1995): MAS have been studied, extended, and implemented from both a theoretical and a practical viewpoint. Thanks to a large community and to the interest — most of the time — of the industry, MAS have become a good base for the design and development of heterogeneous and distributed systems.

Despite that, the diffusion of MAS applications in the real world has not reached its full potential yet. Although dating back to ten years ago, the work by Pechoucek et al. (Pechoucek and Marík, 2008) explains that the reasons can be found in

1. limited awareness about the potential of agent technologies;
2. limited publicity of successful industrial projects carried out with agent technologies;
3. misunderstandings about the effectiveness of agent-based solutions;
4. risks of adopting a technology that has not been already proven in large scale industrial applications;
5. lack of mature enough design and development tools for industrial deployment.

All the observations previously reported are important per se, and should be tackled individually. The main work presented in this thesis tackles the 4th and 5th issues.

One of the risks of adopting MAS technologies for the development of large real systems is related to their reliability. MAS are essentially distributed systems composed of **decentralized** autonomous entities called agents[1]. Each agent is able to reason about its goals in order to achieve them, usually collaborating in a community with other agents, exploiting social activities. The importance given to the features of the agents may change with respect to the scenario they have been exploited in. Features such as autonomy are commonly recognized to be important for an agent. However, when environments are highly dynamic as in modern embedded systems (Macías-Escrivá et al., 2013), where the agents have to make decisions on adaptivity at runtime with respect to changing requirements, self-adaptivity is as important as autonomy; on the other hand, when the environment where agents operate is almost static, self-adaptivity can be an undesired feature. As an example, we can think

[1] Further details can be found in Chapter 3.

about developing a MAS supporting a flight control system. In such scenario, we do not want agents able to adapt, we want agents that can be trusted and, above all, with a predictable behaviour.

Taking care of the complexity of such kind of distributed systems is not an easy task. The complexity easily raises design and implementation errors, and this causes a major risk in adopting MAS technology in large scale industrial applications. One of the possible ways to tackle this kind of problem is through verification of MAS. Said verification – showing that a system is correct with respect to its stated requirements – is an increasingly important issue, especially as agent systems are applied to safety-critical applications such as autonomous spacecraft control (Havelund, Lowry, and Penix, 2001; Muscettola et al., 1998). By verifying the behaviours of the agents, we increase the level of trust in them. As we are going to analyze more in detail in the thesis, we will discuss about different approaches to the verification of MAS. In this thesis, we will focus more on the Runtime Verification (RV) approach, but we will also consider its possible combination with the more standard Static Verification approach (Fisher and Wooldridge, 1997) such as model checking (Bordini et al., 2006; Clarke, Grumberg, and Peled, 1999; Merz, 2001). Being able to verify the behaviour of the agents inside a MAS increases the reliability of the system implementation. If we could check the MAS in respect of a set of properties of our interest, we would also be able to use it in scenarios where reliability is a key feature. For this reason, we focused our attention on studying and developing new approaches to achieve the RV of MAS during the Ph.D. program.

Runtime Verification (Delgado, Gates, and Roach, 2004; Leucker and Schallhart, 2009) is a light-weight approach that allows us to verify if a software system is consistent or not with a specific property. The name “Runtime” derives from the time this approach is applied. In RV, we check a system while it is running; thanks to this, we can state that RV is a light-weight approach because it does not care about generating all possible system’s behaviours (as it usually happens in model checking) but it simply analyzes the observed ones’[2]. For this reason, RV is also commonly associated with the term “Monitoring” because we are just interested in verifying what the system does (obtaining a less invasive but non exhaustive approach). The formalisms that we can use to represent these properties can vary, and each one has its advantages and disadvantages. In Chapter 5 we present different ones that can be used to specify protocols and properties; in Chapter 16 we compare them to the one we are going to present and use in this thesis. More specifically, we chose the Trace Expression formalism, which is presented in Chapter 4, to specify properties. Trace expressions are a complex and compact formalism that has been developed during the Ph.D. program. Despite showing how this formalism can actually be used to achieve the RV of MAS, we will also present the extensions that have been proposed during the Ph.D. program in Chapters 6 and 7

[2]In an ideal world, the monitor should never influence the system. In a nutshell, the presence of the monitor does not change the system’s behaviour.

in order to increase the range of its possible applications. RV reinforces the reliability of the MAS introducing monitors in the implementations. Through these monitors, we can analyze at runtime the behaviour of all the agents and we can check if it is consistent with a specification. For instance, as we will see in Chapter 8, we can specify Agent Interaction Protocols (AIP) in order to check the social activities of the agents. In this kind of situation, the monitors must be able to observe the message exchanges among the agents checking the compliance with the AIP at runtime. This is interesting when we consider MAS where the social activities are one of the main aspects. Monitors can also be used to check the internal states of the agents, but we will always focus on a less invasive verification approach in this work; as a matter of fact, an entire part of the thesis is centred on the use of trace expressions to represent AIPs. Outside the context of AIPs, scenarios where we have no access to the implementations of the systems are common, and the possibility to analyze them externally is an important and valuable feature.

In order to exhaustively achieve the RV of a software system, we have to tackle it from both a theoretical and a practical point of view. Theoretical view, because RV is a formal approach based on the concept of formal specifications for representing the properties used to guide the monitors; thus, we need to study, integrate and extend new formalisms to support the verification process. Practical view, because as the other verification approaches, its aim is to verify real systems in real scenarios; works on RV and Static Verification are, in fact, usually accompanied with their implementation and related experiments. This aspect is well established in this thesis, an entire Chapter is indeed dedicated to the implementation of a tool supporting our RV approach (Chapter 14).

A standard way to achieve the RV of a MAS is using a single centralized monitor that is able to observe the behaviours of all the agents belonging to the MAS. This approach is extremely monolithic and is generally suitable for more centralized systems. If we consider a large MAS, composed by many different agents that are spread heterogeneously inside an environment, verifying the behaviours of all of them using only one monitor becomes easily intractable (typical bottleneck problem). For this reason, during the Ph.D. program, we have been focusing on a more decentralized way to achieve the RV of MAS, presented in Chapter 9. Intuitively, we are interested in decentralizing the monitoring of specifications on multiple decentralized monitors. Instead of generating a single centralized monitor to check the entire system behaviour, we will show how we can achieve the same results producing different decentralized monitors. Not all the specifications expressed using trace expressions can be decentralized in the same way and there are many issues to be tackled in order to achieve this.

As we anticipated before, RV is not the only way to increase the reliability of MAS. During the Ph.D. program, we also studied possible ways to combine static verification and runtime verification, presented in Chapter 12 and Chapter 13. Studying new hybrid approaches is very important because static and runtime verifications have different advantages and disadvantages, as we will

see, and their combination produces a more robust way to verify MAS. For this and other reasons, we dedicated part of the Ph.D. program in studying two different hybrid approaches.

At the beginning of this chapter, we presented a list of possible issues concerning the use of MAS for developing real life applications. Until now we have considered the parts of the thesis focused on reducing the risks of adopting agent technology in real applications. The other relevant claim tackled by the thesis concerns the development of tools supporting the design and development of MAS, particularly the development of a tool supporting the RV of MAS. From a software engineering viewpoint, proposing new techniques and methods for achieving the RV of MAS is useless if they are not supported by a suitable development toolkit. In the last year of the Ph.D. program we developed an Integrated Development Environment (IDE), presented in Chapter 14, that supports the definition of specifications and the automatic generation of the corresponding monitors for achieving the RV of the target MAS.

The main objectives of this thesis can be summarized as follows:

- to present the **theoretical** foundations of the trace expression formalism and its extensions (*parametric* and *probabilistic*);
- to show how to exploit the trace expression formalism for achieving the **Runtime Verification** of – and not limited only to – distributed artificial intelligence systems (such as Multiagent Systems);
- to present the algorithms that have been designed and developed to obtain a **decentralized** Runtime Verification of Agent Interaction Protocols in the context of verification of the social activities involved in the Multiagent Systems (also in presence of uncertainty in the observed events);
- to **combine** Runtime Verification and Static Verification in both directions in order to obtain a more robust and reliable approach for the verification of the Multiagent Systems;
- to **implement** a software platform for supporting the use of the trace expression formalism for the design and development of monitors for verifying Multiagent Systems at runtime.

The research methodology followed during the three years of the Ph.D. program consisted in facing every research issue both from a theoretical and practical viewpoint.

2 Structure and Contributions of the Thesis

2.1 Structure

PART I - INTRODUCTION. This part contains introductory chapters explaining the motivations and aims of this thesis (Chapter 1). The structure of the thesis is presented here in this chapter.

PART II - BACKGROUND. We introduce all the preliminary concepts necessary to understand the contents of the thesis (Chapter 3). In this part, we briefly present the concepts of MAS, Runtime Verification and Static Verification. We also present in detail the Linear Temporal Logic (LTL), which is used several times in the thesis. In Chapter 4 we define the base formalism we use in the entire work – the thesis is also named after it: the trace expression formalism. Its original version has been defined before starting the Ph.D. program inside my master's thesis. For this reason the canonical version is not considered a novel part of the thesis and is listed here in Part II. In Chapter 5 we present the general state of the art analysis while reserving a more specific study for each single chapter.

PART III - FORMALISM EXTENSIONS. As anticipated before, since our objective is to verify software systems – in particular MAS – it was natural to study and propose new extensions for our formalism. In particular, we proposed parametric (Chapter 6) and probabilistic (Chapter 7) extensions. Each extension has its advantages and disadvantages and is presented separately with its motivations and examples.

PART IV - ENGINEERING AGENT INTERACTION PROTOCOLS WITH TRACE EXPRESSIONS. It is the most important part of the entire work. In this part we present how to specify Agent Interaction Protocols (Chapter 8) using the trace expression formalism and we focus on the resulting issues when these are used to achieve decentralized RV (Chapter 9 and Chapter 10) and protocol conformance analysis (Chapter 11).

PART V - COMBINING STATIC AND RUNTIME VERIFICATION. RV of MAS is interesting, but even more so when combined with static verification. During the Ph.D. program we combined our approach with existing static verification ones. In Chapter 12, we show how to model check our specifications – trace expressions – with respect to a given LTL property. In Chapter 13, we instead present a way to simplify the model checking of MAS using RV. More specifically, we show how we can make assumptions on the model in order to

reduce the state space explosion during the static verification phase and how to check that these assumptions still hold at runtime.

PART VI - IMPLEMENTATION AND CASE STUDY. We dedicated an entire part of the thesis to the presentation of the IDE that has been developed for supporting the trace expression formalism. In this part we also show examples of its use through screenshots of the resulting implemented plugin while a brief tutorial will be given at the end. The second chapter is instead dedicated to presenting a challenging case study where we show how to use our formalism for representing medical guidelines.

PART VII - DISCUSSION. At the beginning of the thesis we present the state of the art of our work. In particular, we present different other formalisms that are close to ours. In this part, we focus on the main differences with respect to the trace expression formalism and present a comparison table listing them (Chapter 16). In Chapter 17 we end with the conclusions and future work.

This thesis can be further divided in three macro areas:

- Part I, Part III and Part IV concern the most theoretical aspects of this thesis. This is also the macro area of the thesis where all the theoretical and technical background is presented as well as the algorithms for MAS decentralized runtime verification.
- Part V focuses on the hybrid combination of static and runtime verification. This is the part of the thesis that is more focused on the applied aspects.
- Part VI presents the practical aspects concerning the development of the tool supporting our runtime verification approach, and a case study.

2.2 Contributions

We can classify the contributions of the thesis into four categories. The first two are related to the study of new formalism extensions to increase the range of possible uses, and of algorithms to achieve the decentralization of specifications in order to obtain a decentralized runtime verification approach (extremely important in the context of multiagent systems). The last two are instead focused more on applicative and implementation aspects (hybrid combination of runtime and static verification, and the development of a platform supporting our verification approach).

In the following paragraphs we summarize the four main contributions of this thesis.

2.2.1 Propose an expressive formalism and its extensions to support runtime verification of complex systems

In Chapters 6 and 7 we present the latest advances in the extension of the trace expression formalism. Both these extensions allow the formalism to be used in a larger set of scenarios. The parametric extension is useful to represent data, objects, time and generic values within specifications. Thanks to this extension, we can represent more complex systems and model more realistic agent interaction protocols for the verification of the social activities of the agents inside a multiagent system. The probabilistic extension is useful in scenarios where we can have uncertainty in the observed events. Considering the multiagent system scenario, this is extremely important when we are interested in analyzing communicative aspects where messages can be lost or there can be issues on the communication channels. Updating the runtime verification approach to support the absence of information is extremely useful and makes our approach reliable because, even though we do not have the complete trace produced by the system, we can still conclude with a certain probability that the system satisfies a given interaction protocol.

Chapter 6 is based on the following publications:

- Davide Ancona, Angelo Ferrando, and Viviana Mascardi (2016). “Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday”. In: Cham: Springer International Publishing. Chap. Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification, pp. 47–64. ISBN: 978-3-319-30734-3
- Davide Ancona et al. (2017a). “Parametric Trace Expressions for Runtime Verification of Java-Like Programs”. In: *Proc. of the 19th Workshop on Formal Techniques for Java-like Programs*. ACM, 10:1–10:6. DOI: 10.1145/3103111.3104037. URL: http://doi.acm.org/10.1145/3103111.3104037
- D. Ancona, A. Ferrando, and V. Mascardi (2017). “Parametric Runtime Verification of Multiagent Systems (extended abstract)”. In: *Proc. of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017*. Ed. by Kate Larson et al. ACM, pp. 1457–1459

2.2.2 Design and define the algorithms for obtaining a decentralized runtime verification approach of Agent Interaction Protocols

In Chapters 8, 9 and 10 we present how to define Agent Interaction Protocols and how to decentralize their runtime verification process with or without uncertainty. This part is more focused on the presentation of the algorithms supporting this kind of decentralization. Most importantly, all the issues related to the representation of Agent Interaction Protocols when we want to decentralize the monitoring of multiagent system are presented and tackled in this part of the thesis. In Chapter 10, the decentralization study is also made

more difficult and innovative from the presence of gaps in the events observed by distributed monitors.

The decentralization of agent interaction protocols also brings us to the reuse of agents. Once we know that an agent is compliant to a protocol, as long as we are able to assert that this protocol is *conformant* to another one, we can find ourselves in a situation where we can reuse the agent inside a different protocol. In Chapter 11, the thesis presents and analyzes a kind of conformance test which allows reusing agents even when the alphabets used inside the protocols are different.

Chapter 8 is based on the following publication:

- Davide Ancona et al. (2018a). "Coping with Bad Agent Interaction Protocols When Monitoring Partially Observable Multiagent Systems". In: *PAAMS*. Vol. 10978. Lecture Notes in Computer Science. Springer, pp. 59–71

Chapter 9 is based on the following publications:

- Angelo Ferrando, Davide Ancona, and Viviana Mascardi (2017). "Decentralizing MAS Monitoring with DecAMon". In: *Proc. of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017*. Ed. by Kate Larson et al. ACM, pp. 239–248. URL: `http://dl.acm.org/citation.cfm?id=3091164`
- Angelo Ferrando (2016). "Automatic Partitions Extraction to Distribute the Runtime Verification of a Global Specification". In: *Proceedings of the Doctoral Consortium of AI*IA 2016 co-located with the 15th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2016), Genova, Italy, November 29, 2016*. Ed. by Viviana Mascardi and Ilaria Torre. Vol. 1769. CEUR Workshop Proceedings. CEUR-WS.org, pp. 40–45. URL: `http://ceur-ws.org/Vol-1769/paper07.pdf`
- Davide Ancona et al. (2016). "MAS-DRiVe: a Practical Approach to Decentralized Runtime Verification of Agent Interaction Protocols". In: *Proc. of the 17th Workshop "From Objects to Agents"*. Ed. by Corrado Santoro, Fabrizio Messina, and Massimiliano De Benedetti. Vol. 1664. CEUR Workshop Proceedings. CEUR-WS.org, pp. 35–43. URL: `http://ceur-ws.org/Vol-1664/w7.pdf`

Chapter 11 is based on the following publication:

- Davide Ancona, Angelo Ferrando, and Viviana Mascardi (2018a). "Agents Interoperability via Conformance Modulo Mapping". In: *Proceedings of the 19th Workshop "From Objects to Agents", Palermo, Italy, June 28-29, 2018*. Ed. by Massimo Cossentino, Luca Sabatucci, and Valeria Seidita. Vol. 2215. CEUR Workshop Proceedings. CEUR-WS.org, pp. 109–115. URL: `http://ceur-ws.org/Vol-2215/paper_18.pdf`

2.2.3 Propose a hybrid approach combining runtime and static verification of multiagent systems

In Chapters 12 and 13 we present two different ways to combine model checking and runtime verification through trace expressions. In both scenarios the aim is to create a more robust and reliable verification approach since model checking and runtime verification have disadvantages which can be solved by their combination. During the Ph.D. program, we studied many different approaches in both research fields but we only found few proposals that combine both of them. Some work in this direction can be found in (Ahrendt, Pace, and Schneider, 2016; Ahrendt et al., 2016; Artho and Biere, 2005; Artho et al., 2004; Chimento et al., 2015; Colombo, Pace, and Schneider, 2009; Gui et al., 2013).

Chapter 12 is based on the following publication:

- Angelo Ferrando (2019). “The early bird catches the worm: First verify, then monitor!” In: *Science of Computer Programming* 172, pp. 160 –179. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2018.11.008`. URL: `http://www.sciencedirect.com/science/article/pii/S0167642318304349`

Chapter 13 is based on the following publication:

- Angelo Ferrando et al. (2018a). “Recognising Assumption Violations in Autonomous Systems Verification”. In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*. Ed. by Elisabeth André et al. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, pp. 1933–1935. URL: `http://dl.acm.org/citation.cfm?id=3238028`

2.2.4 Present a tool developed for supporting runtime verification guided by trace expressions and a case study

In Chapter 14 we present the tool which has been developed for supporting the trace expression formalism and the automatic generation of decentralized monitors for achieving the decentralized runtime verification of multiagent systems implemented using the JADE framework (Bellifemine, Caire, and Greenwood, 2007).

Chapter 14 is based on the following publications:

- Angelo Ferrando (2017). “RIVERtools: an IDE for RuntIme VERification of MASs, and Beyond”. In: *PRIMA Demo Track 2017*
- Davide Ancona et al. (2018b). “Managing Bad AIPs with RIVERtools”. In: *PAAMS*. Vol. 10978. Lecture Notes in Computer Science. Springer, pp. 296–300

In Chapter 15 we propose a case study in the field of Remote Patient Monitoring. In this case study we show how to use our formalism to formally specify medical guidelines.

Chapter 15 is based on the following publications:

- Angelo Ferrando, Davide Ancona, and Viviana Mascardi (2016). "Monitoring Patients with Hypoglycemia Using Self-adaptive Protocol-Driven Agents: A Case Study". In: *Proc. of Engineering Multi-Agent Systems - 4th International Workshop, EMAS 2016, Revised, Selected, and Invited Papers.* Ed. by Matteo Baldoni et al. Vol. 10093. LNCS. Springer, pp. 39–58. DOI: `10.1007/978-3-319-50983-9_3`. URL: `http://dx.doi.org/10.1007/978-3-319-50983-9_3`
- Davide Ancona, Angelo Ferrando, and Viviana Mascardi (2018b). "Improving flexibility and dependability of remote patient monitoring with agent-oriented approaches". In: *Int. J. Agent-Oriented Software Engineering* 6, Nos. 3/4

2.3 How to read the Thesis

In order to make the reading of the thesis easier, we decided to define the following reading sequences of the main chapters essential to understand the contents according to reader's interest:

- **Chapter sequence for the trace expression formalism:**
  [*Interests: Programming Languages*]
  - **Part II - Background:** 4.
  - **Part III - Formalism extensions:** 6 and 7.
- **Chapter sequence for Agent Interaction Protocols modelling:**
  [*Interests: multiagent systems, Agent Interaction Protocols*]
  - **Part II - Background:** 4.
  - **Part IV - Engineering Agent Interaction Protocols:** 8 and 11.
- **Chapter sequence for decentralized runtime verification:**
  [*Interests: multiagent systems, (decentralized) runtime verification, agent interaction protocols*]
  - **Part II - Background:** 4.
  - **Part IV - Engineering Agent Interaction Protocols:** 8 and 9.
  - **Part VI - Implementation and Case Study:** 14.
- **Chapter sequence for probabilistic runtime verification (*runtime verification with state estimation*):**
  [*Interests: multiagent systems, probabilistic (decentralized) runtime verification, agent interaction protocols*]

  - **Part II - Background:** 4.
  - **Part III - Formalism extensions:** 7.
  - **Part IV - Engineering Agent Interaction Protocol:** 8 and 10.

- **Chapter sequence for hybrid combination of runtime and static verification:**
  [*Interests: multiagent systems, runtime verification, model checking*]
  - **Part II - Background:** 4.
  - **Part V - Combining static and runtime verification:** 12 and 13.

- **Chapter sequence for implementation of runtime verification of Agent Interaction Protocols:**
  [*Interests: multiagent systems, agent interaction protocols, tool implementation, Integrated Development Environment design*]
  - **Part II - Background:** 4.
  - **Part III - Formalism extensions:** (*optional*) 6.
  - **Part IV - Engineering Agent Interaction Protocol:** 8 and (*optional*) 10.
  - **Part VI - Implementation and Case Study:** 14 and 15.

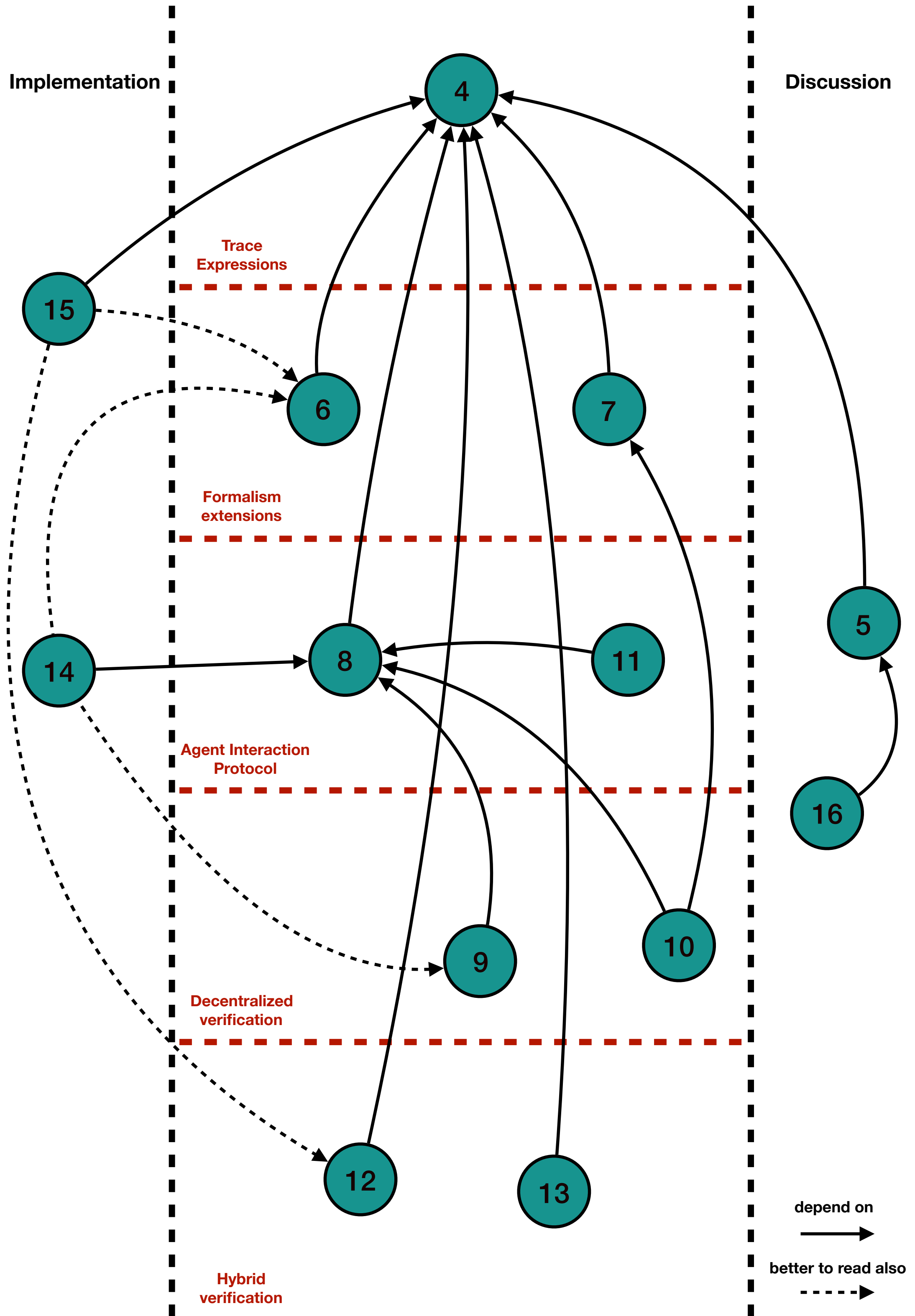

Figure 2.1. Dependency graph of the main chapters. For instance, to understand Chapter 10 you have to read chapters 8 and 7 first.

Part II

Background

This part of the thesis has three different objectives:

- to give a general background for all the contents that might be useful for the reader to understand the contribution of the thesis;
- to present the trace expression formalism, that is the pillar of the thesis and on which all the Ph.D. program has been built;
- to present a general state of the art of the work.

3 *Preliminaries*

"Knowing yourself is the beginning of all wisdom."
- Aristotle

In this chapter, we present all the preliminary concepts that are necessary to understand the contents of this thesis. A reader who is already familiar with the notion of multiagent systems, runtime and static verification, LTL and derivates can skip this chapter and start reading Chapter 4.

3.1 Multiagent Systems and Distributed Artificial Intelligence

Since its inception in the mid to late 1970s distributed artificial intelligence (DAI) evolved and diversified rapidly. Today it is an established and promising research and application field which brings together and draws on results, concepts, and ideas from many disciplines, including artificial intelligence (AI), computer science, sociology, economics, organization and management science, and philosophy.

As defined in (Weiss, 1999), an agent is a computational entity such as a software program or a robot that can be viewed as perceiving and acting upon its environment and that is autonomous in that its behaviour at least partially depends on its own experience. As an intelligent entity, an agent operates flexibly and rationally in a variety of environmental circumstances given its perceptual and effectual equipment. Behavioral flexibility and rationality are achieved by an agent on the basis of key processes such as problem solving, planning, decision making, and learning. As an interacting entity, an agent can be affected in its activities by other agents and perhaps by humans. A key pattern of event in multiagent systems is *goal-* and *task-oriented* coordination, both in cooperative and in competitive situations. In the case of cooperation several agents try to combine their efforts to accomplish as a group what the individuals cannot, and in the case of competition several agents try to get what only some of them can have. The long-term goal of DAI is to develop mechanisms and methods that enable agents to interact like humans (or even better), and to understand events among intelligent entities whether they are computational, human, or both. This goal raises a number of challenging issues that all are centered around the elementary question of when and how to interact with whom.

To make the above considerations more concrete, a closer look has to be taken on multiagent systems and thus on "interacting, intelligent agents":

- "Agents" are autonomous, computational entities that can be viewed as perceiving their environment through sensors and acting upon their environment through effectors. Since they are computational entities, agents must exist in the form of programs that run on computing devices; being autonomous means that to some extent they have control over their behaviour without the intervention of humans and other systems. Agents pursue goals or carry out tasks in order to meet their design objectives, and in general these goals and tasks can be complementary as well as conflicting.
  For N. R. Jennings et al. (Jennings, Sycara, and Wooldridge, 1998), an agent is a computer system, situated in some environment, that is capable of flexible autonomous actions in order to meet its design objectives. There are thus three key concepts in their definition: situatedness, autonomy, and flexibility. In detail,
  - Situatedness, in this context, means that the agent receives sensory input from its environment and that it can perform actions which

change the environment in some way;

- Autonomy is a difficult concept to pin down precisely, but they mean it in the sense that the system should be able to act without the direct intervention of humans (or other agents), and that it should have control over its own actions and internal state;

- By flexible, they mean that the system is: *responsive*, agents should perceive their environment and respond in a timely fashion to changes that occur in it, *pro-active*, agents should not simply act in response to their environment, they should be able to exhibit opportunistic, goal-directed behaviour and take the initiative where appropriate and *social*, agents should be able to interact, when appropriate, with other artificial agents and humans in order to complete their own problem solving and to help others with their activities.

- "Intelligent" indicates that the agents pursue their goals and execute their tasks such that they optimize some given performance measures. To say that agents are intelligent does not mean that they are omniscient or omnipotent, nor does it mean that they never fail. Rather, it means that they operate flexibly and rationally in a variety of environmental circumstances, given the information they have and their perceptual and effectual capabilities.

- "Interacting" indicates that the agents may be affected by other agents or perhaps by humans in pursuing their goals and executing their tasks. Events can take place indirectly through the environment in which they are embedded (e.g., by observing one another or by carrying out an action that modifies the environmental state) or directly through a shared language (e.g., by providing information in which other agents are interested or which confuses other agents). DAI primarily focuses on coordination as a form of event that is particularly important with respect to goal attainment and task completion. The purpose of coordination is to achieve or avoid states of affairs that are considered as desirable or undesirable by one or several agents. To coordinate their goals and tasks, agents have to explicitly take dependencies among their activities into consideration. Two basic, contrasting patterns of coordination are cooperation and competition. In the case of cooperation, several agents work together and draw on the broad collection of their knowledge and capabilities to achieve a common goal. Against that, in the case of competition, several agents work against each other because their goals are conflicting. Cooperating agents try to accomplish as a team what the individuals cannot, and so fail or succeed together. Competitive agents try to maximize their own benefit at the expense of others, and so the success of one implies the failure of others.

Agent Communications Language performatives

Performativity[1] is a term for the capacity of speech and communication not just to communicate but rather to act or consummate an action, or to construct and perform an identity. A common example is the act of saying “I pronounce you man and wife” by a licensed minister before two people who are prepared to wed (or “I do” by one of those people upon being asked whether they take their partner in marriage). An umpire calling a strike, a judge pronouncing a verdict, or a union boss declaring a strike are all examples of performative speech.

Speech Act Theory as introduced by Oxford philosopher J.L. Austin (Austin, 1962) and further developed by American philosopher J.R. Searle, considers the types of acts that utterances can be said to perform:

- Locutionary Acts, the performance of an utterance;
- Illocutionary Acts, the pragmatic ’illocutionary force’ of the utterance;
- Perlocutionary Acts, the actual effect.

Agent Communication Languages (ACLs) are based on the speech act theory: messages are actions, or communicative acts, as they are intended to perform some action by virtue of being sent. The specification consists of a set of message types and the description of their pragmatics, that is the effects on the mental attitudes of the sender and receiver agents. Every communicative act is described with both a narrative form and a formal semantics based on modal logic.

The most popular ACLs are:

- FIPA-ACL[2] (by the Foundation for Intelligent Physical Agents, a standardization consortium);
- KQML (Finin et al., 1994) (Knowledge Query and Manipulation Language).

Both rely on the speech act theory developed by Searle in the 1960s (Searle, 1969) and enhanced by Winograd and Flores in the 1970s. They define a set of performatives, also called Communicative Acts, and their meaning (e.g. tell). The content of the performative is not standardized, but varies from language to language.

FIPA was originally formed as a Swiss based organization in 1996 to produce software standards specifications for heterogeneous and interacting agents and agent based systems. Since its foundations, FIPA has played a crucial role in the development of agents standards and has promoted a number of initiatives and events that contributed to the development and uptake of agent technology.

[1] https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Performativity.html

[2] http://www.fipa.org/specs/fipa00001/, http://www.fipa.org/specs/fipa00007/, http://www.fipa.org/specs/fipa00025/, http://www.fipa.org/specs/fipa00037/

KQML is part of the ARPA Knowledge Sharing Effort, which is a consortium to develop conventions facilitating the sharing and reuse of knowledge bases and knowledge-based systems. Its goal is to define, develop, and test infrastructure and supporting technology to enable participants to build much bigger and more broadly functional systems than could be achieved working alone.

Speech act theory uses the term performative to identify the illocutionary force of this special class of utterance. Example performative verbs include promise, report, convince, insist, tell, request, and demand. Illocutionary forces can be broadly classified as assertives (statements of fact), directives (commands in a master-slave structure), commissives (commitments), declaratives (statements of fact), and expressives (expressions of emotion). Performatives are usually represented in the stylized syntactic form "I hereby tell..." or "I hereby request..." Because performatives have the special property that "saying it makes it so," not all verbs are performatives. For example, stating that "I hereby solve this problem" does not create the solution. Although the term speech is used in this discussion, speech acts have to do with communication in forms other than the spoken word. In summary, speech act theory helps define the type of message by using the concept of the illocutionary force, which constrains the semantics of the communication act itself. The sender's intended communication act is clearly defined, and the receiver has no doubt as to the type of message sent. This constraint simplifies the design of software agents.

To make agents understand each other they have to speak the same language; they can also have a common ontology.

In 1993, Gruber originally defined the notion of an ontology as an "explicit specification of a conceptualization" (Gruber, 1993). In 1997, Borst defined an ontology as a "formal specification of a shared conceptualization" (Borst, 1997). In 2007, Gruber revised his definition of ontology; in the context of computer and information sciences, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application (Gruber, 2009). We present the notion of ontology here because ontologies are usually part of the agent's knowledge base and they are used to describe what kind of things an agent can deal with and how they are related to each other. The connection between ontology, semantic web and MAS is not new, and in literature we can found different works showing the advantages in the combination of these two research areas[3] (Hendler, 2001; Mascardi et al., 2011; Moreira et al., 2006). Both ontology and agent technologies are central to the semantic web, and their combined use will enable the sharing of heterogeneous, autonomous

[3] http://www.obitko.com/tutorials/ontologies-semantic-web/fipa-ontology-service.html

knowledge sources in a capable, adaptable and extensible manner. Ontology is used throughout the MAS to assist the interactions among different agents as well as to improve the quality of the service provided by each agent.

3.2 Rational agents

The key aspect of a rational agent (Bratman, 1987; Cohen and Levesque, 1990; Rao and Georgeff, 1992) is that the decisions it makes, based on dynamic motivations, should be both "reasonable" and "justifiable". One of the most famous and used model used for defining rational agents is called belief-desire-intention (BDI) model. Belief-desire-intention architectures originated from the work of the Rational Agency project at Stanford Research Institute in the mid-1980s. The origins of the model lie in the theory of human practical reasoning developed by the philosopher Michael Bratman (Bratman, 1987), which focuses particularly on the role of intentions in practical reasoning. The conceptual framework of the BDI model is described in (Bratman, Israel, and Pollack, 1988), which also describes a specific BDI agent architecture called IRMA. In detail, a BDI agent comprises *beliefs* that the agent has about itself and its environment, *desires* (or *goals*) representing its long-term aims, and *intentions* describing the agent's immediate goals (the ones it is currently trying to achieve through acting on the environment where it is situated). Using this new notion of rational agent, we can rethink the multiagent system as a system consisting of a number of rational agents interacting with each other (naturally it is not always necessary to have rational agents inside multiagent systems). The cooperation aspect helps to solve problems that are difficult to solve by individual agents or traditional computer systems. As we are going to discuss deeply in this thesis, the social aspects are crucial for a multiagent systems and require to be well defined and verified in order to make the implementations more robust and reliable.

In the following section, we are going to briefly introduce one of the most famous and used engine for developing BDI agents. In particular, this engine exploits AgentSpeak (Rao, 1996) as agent-oriented programming language.

3.3 Jason

Jason[4] (Bordini, Hübner, and Wooldridge, 2007) is an engine for an extended version of the AgentSpeak language (Rao, 1996). It implements the operational semantics of that language, and provides a platform for the development of multiagent systems, with many user-customisable features. Jason is available Open Source, and is distributed under GNU LGPL.

One of the most interesting aspects of AgentSpeak is that it was inspired by and based on a model of human behaviour that was developed by philosophers, the BDI model.

We mentioned here Jason because is one of the most famous and used agent platform supporting the BDI model. Besides AgentSpeak, other languages

[4] `http://jason.sourceforge.net/`

developed for programming rational agents include 3APL (Dastani, Riemsdijk, and Meyer, 2005; Hindriks et al., 1997), DRIBBLE (Riemsdijk, Hoek, and Meyer, 2003), Jadex (Pokahr, Braubach, and Lamersdorf, 2005), GOAL (Boer et al., 2007; Hindriks et al., 2000), CAN (Winikoff et al., 2002), SAAPL (Winikoff, 2007), GWENDOLEN (Dennis and Farwer, 2008), and METATEM (Fisher and Ghidini, 2010; Fisher and Hepple, 2009).

3.4 *JADE*

JADE[5] (Java Agent DEvelopment Framework) (Bellifemine, Caire, and Greenwood, 2007) is a software Framework fully implemented in the Java language. It simplifies the implementation of multiagent systems through a middleware that complies with the FIPA specifications[6] and through a set of graphical tools that support the debugging and deployment phases. A JADE-based system can be distributed across machines (which not even need to share the same OS) and the configuration can be controlled via a remote GUI. The configuration can be even changed at run-time by moving agents from one machine to another, as and when required. JADE is completely implemented in Java language and the minimal system requirement is the version 5 of JAVA (the run time environment or the JDK).

Besides the agent abstraction, JADE provides a simple yet powerful task execution and composition model, peer to peer agent communication based on the asynchronous message passing paradigm, a yellow pages service supporting publish subscribe discovery mechanism and many other advanced features that facilitate the development of a distributed system.

Thanks to the contribution of the LEAP project[7], ad hoc versions of JADE exist designed to deploy JADE agents transparently on different Java-oriented environments such as Android devices and J2ME-CLDC MIDP 1.0 devices[8]. Furthermore suitable configurations can be specified to run JADE agents in networks characterized by partial connectivity including NAT and firewalls as well as intermittent coverage and IP-address changes.

3.4.1 Architecture

This and the next section on JADE architecture are taken from the tutorial[9] and the book (Bellifemine, Caire, and Greenwood, 2007) with no or limited changes, and are included in this Ph.D. thesis to make it self-contained.

[5] `http://jade.tilab.com/`

[6] `http://www.fipa.org/specs/fipa00001/`, `http://www.fipa.org/specs/fipa00007/`, `http://www.fipa.org/specs/fipa00025/`, `http://www.fipa.org/specs/fipa00037/`

[7] LEAP is the name of the European IST project that developed the LEAP add-on. The Consortium of the LEAP Project was formed by Motorola, Siemens AG, Telecom Italia, Broadcom, University of Parma and ADAC.

[8] `https://www.oracle.com/technetwork/java/faqs-140539.html`

[9] `http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf`

A JADE platform is composed of agent containers that can be distributed over the network. Agents live in containers which are the Java process that provide the JADE runtime and all the services needed for hosting and executing agents. There is a special container, called the *main container*, which represents the bootstrap point of a platform: it is the first container to be launched and all other containers must join to a main container by registering with it.

Each agent is identified by a globally unique name (the AgentIdentifier, or AID, as defined by FIPA). It can join and leave a host platform at any time and can discover other agents through both white-page and yellow-page services.

When the main container is launched, two special agents are automatically instantiated and started by JADE, whose roles are defined by the FIPA Agent Management standard:

1. The Agent Management System (AMS) is the agent that supervises the entire platform. It is the contact point for all agents that need to interact in order to access the white pages of the platform as well as to manage their life cycle. Every agent is required to register with the AMS (automatically carried out by JADE at agent start-up) in order to obtain a valid `AID`.

2. The Directory Facilitator (DF) is the agent that implements the yellow pages service, used by any agent wishing to register its services or search for other available services. The JADE DF also accepts subscriptions from agents that wish to be notified whenever a service registration or modification is made that matches some specified criteria. Multiple DFs can be started concurrently in order to distribute the yellow pages service across several domains. These DFs can be federated, if required, by establishing cross-registrations with one another which allow the propagation of agent requests across the entire federation.

3.4.2 Creating agents

Creating a JADE agent is as simple as defining a class that extends the `jade.core.Agent` class and implementing the `setup()` method as exemplified in the code below.

```
import jade.core.Agent;
public class HelloWorldAgent extends Agent {
  protected void setup() {
    // Printout a welcome message
    System.out.println("Hello World. I'm an agent!");
  }
}
```

More appropriately a class, such as the `HelloWorldAgent` class shown above, represents a type of agent exactly as a normal Java class represents a type of object. Several instances of the `HelloWorldAgent` class can be launched at runtime. Unlike normal Java objects, which are handled by their references, an agent is always instantiated by the JADE run-time and its reference is never

disclosed outside the agent itself (unless of course the agent does that explicitly). Agents never interact through method calls but rather by exchanging asynchronous messages.

The `setup()` method is intended to include agent initializations. The actual job an agent has to perform is typically carried out within "behaviours". Examples of typical operations that an agent performs in its setup() method are: showing a GUI, opening a connection to a database, registering the services it provides in the yellow pages catalogue and starting the initial behaviours. It is good practice not to define any constructor in an agent class and to perform all initializations inside the `setup()` method. This is because at construction time the agent is not yet linked to the underlying JADE run-time and thus some of the methods inherited from the Agent class may not work properly.

3.4.2.1 Agent identifiers

Consistent with the FIPA specifications, each agent instance is identified by an "agent identifier". In JADE an agent identifier is represented as an instance of the `jade.core.AID` class. The `getAID()` method of the `Agent` class allows retrieval of the local agent identifier. An `AID` object includes a globally unique name (`GUID`) plus a number of addresses. The name in JADE has the form `<local-name>@<platform-name>` such that an agent called `Peter` living on a platform called `foo-platform` will have `Peter@foo-platform` as its globally unique name. The addresses included in the `AID` are the addresses of the platform the agent inhabits. These addresses are only used when an agent needs to communicate with another agent living on a different compliant FIPA platform.

The `AID` class provides methods to retrieve the local name (`getLocalName()`), the `GUID` (`getName()`) and the addresses (`getAllAddresses()`).

We can therefore enrich the welcome message of our `HelloWorldAgent` as follows:

```
protected void setup() {
  // Printout a welcome message
  System.out.println("Hello World. I'm an agent!");
  System.out.println("My local-name is "+getAID().getLocalName());
  System.out.println("My GUID is "+getAID().getName());
  System.out.println("My addresses are:");
  Iterator it = getAID().getAllAddresses();
  while (it.hasNext()) {
    System.out.println("- "+it.next());
  }
}
```

The local name of an agent is assigned at start-up time by the creator and must be unique within the platform. If an agent with the same local name already exists in the platform, the JADE runtime prevents the creation of the new agent.

```
String localname = "Peter";
AID id = new AID(localname, AID.ISLOCALNAME);
```

The platform name is automatically appended to the `GUID` of the newly created `AID` by the JADE runtime. Similarly, knowing the `GUID` of an agent, its `AID` can be obtained as follows:

```
String guid = "Peter@foo-platform";
AID id = new AID(guid, AID.ISGUID);
```

3.4.2.2 Agent initialization

The `HelloWorldAgent` class described previously can be compiled, as with normal Java classes, by typing:

```
javac -classpath <JADE-classes> HelloWorldAgent.java
```

Of course the JADE libraries must be in the Classpath for the compilation to succeed. At that point, in order to execute a Hello World agent, i.e. an instance of the `HelloWorldAgent` class, the JADE runtime must be started and a local name for the agent to execute must be chosen:

```
java -classpath <JADE-classes>;. jade.Boot Peter:HelloWorldAgent
```

This command starts the JADE runtime and tells it to launch an agent whose local name is `Peter` and whose class is `HelloWorldAgent`. Again both the JADE libraries and the `HelloWorldAgent` class must be in the Classpath. As a result of the typed command, the following printouts produced by the Hello World agent should appear.

```
Hello World. I’m an agent!
My local-name is Peter
My GUID is Peter@anduril:1099/JADE
My addresses are:
- http://anduril:7778/acc
```

3.4.2.3 Agent tasks

The actual job, or jobs, an agent has to do is carried out within “behaviours”. A behaviour represents a task that an agent can carry out and is implemented as an object of a class that extends `jade.core.behaviours.Behaviour`. To make an agent execute the task implemented by a behaviour object, the behaviour must be added to the agent by means of the `addBehaviour()` method of the `Agent` class.

Behaviours can be added at any time when an agent starts up (in the `setup()` method) or from within other behaviours. Each class extending `Behaviour` must implement two abstract methods. The `action()` method defines the operations to be performed when the behaviour is in execution. The `done()` method returns a boolean value to indicate whether or not a behaviour has completed and is to be removed from the pool of behaviours an agent is executing.

BEHAVIOUR SCHEDULING AND EXECUTION. An agent can execute several behaviours concurrently. However, it is important to note that the scheduling of behaviours in an agent is not pre-emptive (as for Java threads), but cooperative. *This means that when a behaviour is scheduled for execution its `action()` method is called and runs until it returns.* Therefore it is the programmer who defines when an agent switches from the execution of one behaviour to the execution of another.

This approach often creates difficulties for inexperienced JADE developers and must always be kept in mind when writing JADE agents. Though requiring an additional effort, this model does have several advantages:

- It allows a single Java thread per agent which is quite important especially in environments with limited resources such as cellphones.
- It provides improved performance since behaviour switching is far faster than Java thread switching.
- It eliminates all synchronization issues between concurrent behaviours accessing the same resources since all behaviours are executed by the same Java thread. This also results in a performance enhancement.
- When a behaviour switch occurs, the status of an agent does not include any stack information, implying that it is possible to take a 'snapshot' of it. This allows the implementation of some important advanced features, such as saving the status of an agent in a persistent storage for later resumption (agent persistency), or transferring the agent to another container for remote execution (agent mobility).

It is important to note that a behaviour such as that shown below will prevent any other behaviour from being executed because its `action()` method will never return.

```
public class OverbearingBehaviour extends Behaviour {
  public void action() {
    while (true) {
      // do something
    }
  }
  public boolean done() {
    return true;
  }
}
```

ONE-SHOT BEHAVIOUR, CYCLIC BEHAVIOUR AND GENERIC BEHAVIOURS. The three primary behaviour types available with JADE are as follows:

1. *One-shot* behaviours are designed to complete in one execution phase; their `action()` method is thus executed only once.

The `jade.core.behaviours.OneShotBehaviour` class already implements the done() method by returning true and can be conveniently extended to implement new one-shot behaviours.

```
public class MyOneShotBehaviour extends OneShotBehaviour {
  public void action() {
    // perform operation X
  }
}
```

In this example, operation `X` is performed only once.

2. *Cyclic* behaviours are designed to never complete; their action() method executes the same operations each time it is called.
   The `jade.core.behaviours.CyclicBehaviour` class already implements the `done()` method by returning false and can be conveniently extended to implement new cyclic behaviours.

```
public class MyCyclicBehaviour extends CyclicBehaviour {
  public void action() {
    // perform operation Y
  }
}
```

In this example, operation `Y` is performed repetitively until the agent executing the behaviour terminates.

3. *Generic* behaviours embed a status trigger and execute different operations depending on the status value. They complete when a given condition is met.

```
public class ThreeStepBehaviour extends Behaviour {
  private int step = 0;
  public void action() {
    switch (step) {
      case 0:
        // perform operation X
        step++;
        break;
      case 1:
        // perform operation Y
        step++;
        break;
      case 2:
        // perform operation Z
        step++;
        break;
    }
  }
  public boolean done() {
    return step == 3;
  }
}
```

In this example, the step member variable implements the status of the behaviour. Operations X, Y and Z are performed sequentially after which the behaviour completes.

JADE also provides the possibility of composing behaviours together to create complex behaviours.

SCHEDULING OPERATIONS. JADE provides two ready-made classes (in the `jade.core.behaviours` package) which can be implemented to produce behaviours that execute at selected points in time.

1. The `WakerBehaviour` has `action()` and `done()` methods pre-implemented to execute the `onWake()` abstract method after a given timeout (specified in the constructor) expires. After the execution of the `onWake()` method the behaviour completes.

```
public class MyAgent extends Agent {
  protected void setup() {
    System.out.println("Adding waker behaviour");
    addBehaviour(new WakerBehaviour(this, 10000) {
      protected void onWake() {
        // perform operation X
      }
    });
  }
}
```

   In this example, operation X is performed 10 seconds after the `'Adding waker behaviour'` text is printed.

2. The `TickerBehaviour` has `action()` and `done()` methods pre-implemented to execute the `onTick()` abstract method repetitively, waiting a given period (specified in the constructor) after each execution.
   A `TickerBehaviour` never completes unless it is explicitly removed or its `stop()` method is called.

```
public class MyAgent extends Agent {
  protected void setup() {
    addBehaviour(new TickerBehaviour(this, 10000) {
      protected void onTick() {
        // perform operation Y
      }
    });
  }
}
```

   In this example, operation Y is performed periodically every 10 seconds.

3.5 Techniques for checking the system's behaviour

There are two main approaches for verifying a software system: Static Verification and Runtime Verification. Static Verification includes all that kind

of verification techniques where the code is inspected before it is executed. One of the most famous approaches in this research area is Model Checking (Clarke, Grumberg, and Peled, 1999; Merz, 2001). In a nutshell, Model Checking is a method to algorithmically verify formal systems. Namely, the verification is done on a model that has been derived directly from the real software/hardware one. The properties that are usually checked through these kind of approaches are temporal logic formulas. Conversely, Runtime Verification checks the software/hardware system directly when the system is running. The main difference between the two approaches lies in which types of events they check, and when. Model checking generates the events trying to cover the entire model's behaviour, for this reason is an exhaustive approach (pros) at the expense of performance (cons), while Runtime Verification observes the events when these are generate by the system execution, thus it is not exhaustive because can only check the current observed behaviour (cons), but it presents better performances and is usually a less invasive approach (pros), as we are going to see in the rest of the thesis.

3.5.1 Model checking

Model checking (Clarke, Grumberg, and Peled, 1999) is a technique for verifying finite state concurrent systems such as sequential circuit designs and communication protocols. It has a number of advantages over traditional approaches that are based on simulation, testing, and deductive reasoning. In particular, model checking is automatic and usually quite fast. Also, if the design contains an error, model checking will produce a counterexample that can be used to pinpoint the source of the error. The method, which was awarded the 1998 ACM Paris Kanellakis Award for Theory and Practice, has been used successfully in practice to verify real industrial designs, and companies are beginning to market commercial model checkers.

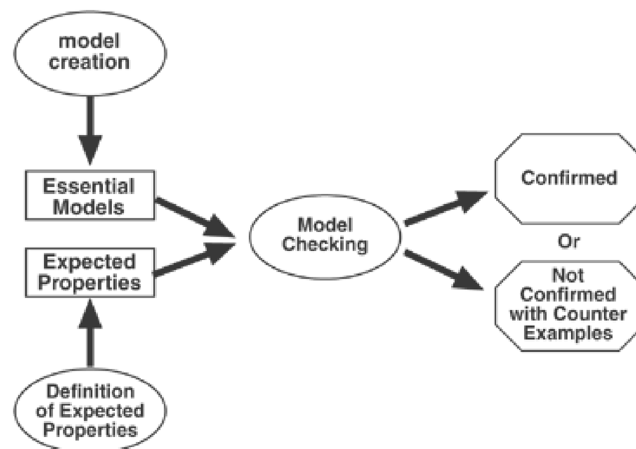

FIGURE 3.1. Model checking procedure (Gluch et al., 2019)

The main challenge in model checking is dealing with the state space explo-

sion problem. This problem occurs in systems with many components that can interact with each other or systems with data structures that can assume many different values. In such cases the number of global states can be enormous. Researchers have made considerable progress on this problem over the last ten years.

Below we report the main advantages and disadvantages in the use of model checking technique (Clarke, 2008).

Main advantages of model checking:

- the user does not need to construct a correctness proof;
- it is fast compared to other rigorous methods such as the use of a proof checker;
- if the specification is not satisfied, the model checker will produce a counterexample execution trace that shows why the specification does not hold;
- it has no problem with partial specifications;
- it supports Temporal Logics which can easily express many of the properties that are needed for reasoning about concurrent systems, this is important because reasoning on the concurrency is often quite subtle and it is difficult to verify all possible cases manually.

Main disadvantages of model checking:

- Writing specifications is hard. This is also true for other verification techniques like automated theorem proving (this also occurs in the runtime verification context).
- State explosion is a major problem. The number of global system states of a concurrent system with many processes or complicated data structures can be enormous. All model checkers suffer from this problem. In fact, the state explosion problem has been the driving force behind much of the research in model checking and the development of new model checkers.

3.5.2 Runtime verification

Runtime verification is being pursued as a lightweight verification technique complementing verification techniques such as model checking and testing and establishes another trade-off point between these forces. One of the main distinguishing features of runtime verification is due to its nature of being performed at runtime, which opens up the possibility to act whenever incorrect behaviour of a software system is detected.

We follow (Delgado, Gates, and Roach, 2004; Leucker and Schallhart, 2009) and define a software failure as a deviation between the observed behaviour and the required behaviour of the software system. A fault is defined as the deviation between the current behaviour and the expected behaviour, which is typically identified by a deviation of the current and the expected state of

the system. A fault might lead to a failure, but not necessarily. An error, on the other hand, is a mistake made by a human that results in a fault and possibly in a failure. According to IEEE ("IEEE Standard for Software Verification and Validation" 2005), verification comprises all techniques suitable for showing that a system satisfies its specification. Traditional verification techniques comprise theorem proving (Bertot and Castran, 2010), model checking (Clarke, Grumberg, and Peled, 1999). Traditional validation techniques comprise testing (Broy et al., 2005a; Myers and Sandler, 2004). A relatively new direction of verification is runtime verification, which manifested itself within the previous years as a lightweight verification technique.

Definition 1. *Runtime verification (Leucker and Schallhart, 2009) is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property.*

In runtime verification (RV) dynamic checking of the correct behaviour of a system can be performed by a monitor which is generated from a formal specification of the properties to be verified.

As happens for formal static verification, RV relies on a high level specification formalism to specify the expected properties of a system; similarly to testing, RV is a lightweight, effective but non exhaustive technique to verify complex properties of a system at runtime.

In contrast to formal static verification and testing, RV offers opportunities for error recovery which make this approach more attractive for the development of reliable software: not only a system can be constantly monitored for its whole lifetime to detect possible misbehavior, but also appropriate handlers can be executed for error recovery.

Main advantages of runtime verification:

- it ensures that the system may be stopped the moment issues are identified in a tractable manner;
- verification is not invasive, the system running should not be affected[10] by the presence of the monitor, this is because the monitor does not need to generate the traces that have to checked (in this way the state explosion problem, which is typical of the static verification, does not happen);
- verification continues beyond system deployment.

Main disadvantages of runtime verification:

- it cannot prevent a wrong execution to take place, as it only verifies actual, already happened, traces of events.

3.5.3 Runtime verification versus Model checking

Runtime verification has its origins in model checking, and, to a certain extend, the key problem of generating monitors is similar to the generation of automata

[10] Adding/Removing the monitor should not influence the system.

in model checking. However, there are also important differences to model checking:

- While in model checking, all executions of a given system are examined to answer whether they satisfy a given correctness property φ, which corresponds to the language inclusion problem, runtime verification deals with the word problem.
- While model checking typically considers infinite traces, runtime verification deals with finite executions - as executions have necessarily to be finite.
- While in model checking a complete model is given allowing to consider arbitrary positions of a trace, runtime verification, especially when dealing with online monitoring, considers finite executions of increasing size. For this, a monitor should be designed to consider executions in an incremental fashion.

These differences make it necessary to adapt the concepts developed in model checking to be applicable in runtime verification. For example, while checking a property in model checking using a kind of backwards search in the model is sometimes a good choice, it should be avoided in online monitoring as this would require, in the worst case, the whole execution trace to be stored for evaluation.

Furthermore, model checking suffers from the state explosion problem, which terms the fact that analyzing all executions of a system is typically been carried out by generating the whole state space of the underlying system, which is often huge. Considering a single run, on the other hand, does usually not yield any memory problems, provided that when monitoring online only a finite history of the execution has to be stored. Last but not least, in online monitoring, the complexity for generating the monitor is typically negligible, as the monitor is often only generated once. However, the complexity of the monitor, i.e. its memory and computation time requirements for checking an execution are of important interest, as the monitor is part of the running system and should not influence the it.

3.6 LTL

As we have mentioned before, usually in verification we are interested in checking a specific kind of properties: *temporal properties*. One of the most used ways to represent these properties is through temporal logic, such as Linear Temporal Logic (LTL) (Pnueli, 1977). LTL is a modal logic which has been introduced for specifying temporal properties of systems; despite its original main application is static verification through model checking, more recently it has been adopted as a specification formalism for RV, and some RV tools support it (Chen and Rosu, 2007; Luo et al., 2014a).

3.6.1 LTL syntax and semantics

Given a finite set of atomic propositions AP, the set of LTL formulas over AP is inductively defined as follows:

- *true* is an LTL formula;
- if $p \in AP$ then p is an LTL formula;
- if φ and ψ are LTL formulas then $\neg\psi$, $\varphi \vee \psi$, $X\psi$, and $\varphi\, U \psi$ are LTL formulas.

Additional operators can be derived in the standard way: $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\varphi \Rightarrow \psi = \neg\varphi \vee \psi$, $F\varphi$ (or $\Diamond\varphi$) $= true\, U \varphi$, and $G\varphi$ (or $\Box\varphi$) $= \neg(true\, U \neg\varphi)$.

Let $\Sigma = 2^{AP}$ be the set of all possible subsets of AP; if $p \in AP$ and $a \in \Sigma$, then p holds in a iff $p \in a$. An LTL model is an infinite trace $w \in \Sigma^{\omega}$; $w(i)$ denotes the element $a \in \Sigma$ at position i in trace w; more formally, if $w = aw'$, then $w(0) = a$, and $w(i) = w'(i-1)$ if $i > 0$.

The semantics of a formula φ depends by the satisfaction relation $w, i \models \varphi$ (w satisfies φ in i) defined as follows:

- $w, i \models p$ iff $p \in w(i)$;
- $w, i \models \neg\phi$ iff $w, i \not\models \phi$;
- $w, i \models \varphi \vee \psi$ iff $w, i \models \varphi$ or $w, i \models \psi$;
- $w, i \models X\varphi$ iff $w, i+1 \models \varphi$ (next operator);
- $w, i \models \varphi\, U \psi$ iff $\exists j \geq 0 \;\; w, j \models \psi$ and $\forall 0 \leq k < j \;\; w, k \models \varphi$ (until operator).

Finally, $w \models \varphi$ (w satisfies φ) holds iff $w, 0 \models \varphi$ holds.

We recall that the set of all models of LTL formulas is the language of star-free ω-regular languages over Σ (Cohen, Perrin, and Pin, 1993).

In order to encode an LTL formula into an equivalent trace expression we exploit the result stating that an LTL formula can be translated into an equivalent non deterministic Büchi automaton (Bauer, Leucker, and Schallhart, 2009).

3.6.2 Non deterministic Büchi Automaton

A Büchi automaton (Büchi, 1990) is a type of ω-automaton which extends a finite automaton to infinite inputs. It accepts an infinite input sequence if there exists a run of the automaton that visits (at least) one of the final states infinitely often.

A (non deterministic) Büchi automaton (NBA) is a tuple $(\Sigma, Q, Q_0, \delta, F)$, where

- Σ is a finite alphabet;
- Q is a finite non-empty set of states;

- $Q_0 \subseteq Q$ is a set of initial states;
- $\delta: Q \times \Sigma \rightarrow 2^Q$ is a transition function;
- $F \subseteq Q$ is a set of accepting states.

A run of an automaton $(\Sigma, Q, Q_0, \delta, F)$ on a word $w \in \Sigma^\omega$ is an infinite trace $\rho = q_0 w(0) q_1 w(1) q_2 \ldots$, s.t. $q_0 \in Q_0$, and for all $i \geq 0$ $q_{i+1} \in \delta(q_i, w(i))$. A run ρ is called accepting iff $Inf(\rho) \cap F \neq \emptyset$, where $Inf(\rho)$ denotes the states visited infinitely often.

3.6.3 LTL Model Checking

After having defined a LTL property φ, we might be interested in knowing if our model of the system M satisfies it. This kind of problem can be formulated in the following way.

Given a model M and a LTL formula φ:

1. all traces of M must satisfy φ;
2. if a trace of M does not satisfy φ we have found a *Counterexample*.

We call Σ_M the set of traces of M and Σ_φ the set of traces that satisfy φ. We check if $\Sigma_M \cap \Sigma_{\neg\varphi} = \emptyset$.

3.6.4 Automata-Based Model Checking

One of the most standard approaches is passing through the generation of a Büchi Automaton. Starting from our model M and the LTL property φ we want to verify on it, we generate the corresponding Büchi Automata. After that, to check if $\Sigma_M \cap \Sigma_{\neg\varphi} = \emptyset$, it is enough to make the product of the two Büchi Automata and search if there exist a trace that belongs to it.

To be more clear, the steps we follow are these.

Given a model M and a LTL formula φ:

1. build the Büchi Automaton $B_{\neg\varphi}$;
2. compute product of M and $B_{\neg\varphi}$;
3. the product accepts the traces of M that are also traces of $B_{\neg\varphi}(\Sigma_M \cap \Sigma_{\neg\varphi})$;
4. if at least one sequence is accepted by the product, then we have found one counterexample.

3.6.5 LTL$_3$

LTL$_3$ is a a three-valued semantics (Bauer, Leucker, and Schallhart, 2009) for LTL formulas, devised to adapt the standard semantics to RV, to correctly consider the limitation that at runtime only finite traces can be checked.

Given a finite trace $\sigma \in \Sigma^*$ of length $|\sigma| = n$, a continuation of σ is an infinite trace $w \in \Sigma^\omega$ s.t. for all $0 \leq i < n$ $w(i) = \sigma(i)$.

Given a finite trace $\sigma \in \Sigma^*$, and an LTL formula φ, the LTL3 semantics of φ, denoted by $\sigma \models_3 \varphi$, is defined as follows:

$$\sigma \models_3 \varphi = \begin{cases} \top & \text{iff } w \models \varphi \text{ for all continuations } w \text{ of } \sigma \\ \bot & \text{iff } w \not\models \varphi \text{ for all continuations } w \text{ of } \sigma \\ ? & \text{iff neither of the two conditions above holds} \end{cases}$$

As an example, let us consider the formula $\varphi = p\,U q$, where $p, q \in AP$; according to the definition above, $\{p\}\{p\}\{q\} \models_3 \varphi = \top$, that is, φ is satisfied by the finite trace $\{p\}\{p\}\{q\}$, and monitoring succeeds; $\{p\}\{p\}\emptyset \models_3 \varphi = \bot$, that is, φ is not satisfied by the finite trace $\{p\}\{p\}\emptyset$, and monitoring fails; finally, $\{p\}\{p\}\{p\} \models_3 \varphi = ?$, that is, at this stage monitoring is inconclusive, and the monitor has to keep monitoring the property expressed by φ. Assuming that $AP = \{p, q\}$, the LTL3 semantics of $p\,U q$ corresponds to the finite state machine (FSM) defined in Figure 12.1, which fully determines the expected behaviour of a monitor for the RV of $p\,U q$.

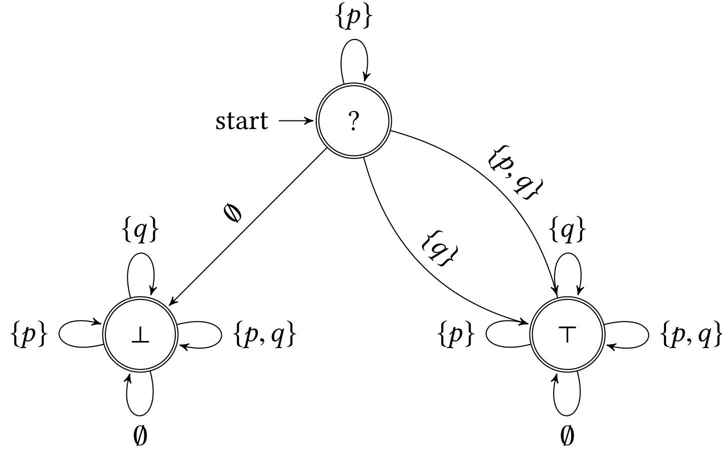

FIGURE 3.2. FSM of the monitor for $p\,U q$, with $AP = \{p, q\}$

More in general, for all LTL formulas φ, it is possible to build an FSM which is a deterministic finite automaton (DFA) where the alphabet is Σ (that is, 2^{AP}), all states are final, each state returns either $\top$ (successful), or $\bot$ (failure), or ? (inconclusive), and the behaviour of the FSM respects the LTL3 semantics of φ: for all finite traces $\sigma \in \Sigma^*$, the FSM accepts σ with final state that returns $v \in \{\top, \bot, ?\}$ iff $\sigma \models_3 \varphi = v$.

The sequence of steps required to generate from an LTL formula φ an FSM that respects the LTL3 semantics of φ is summarized in Figure 3.3.

For each LTL formula φ and $\neg\varphi$ (1), the equivalent NBAs $\mathcal{A}^\varphi$, and $\mathcal{A}^{\neg\varphi}$ are built (2), all states that generate a non empty language are identified (3) and made final and the NBAs are transformed into the corresponding nondeterministic finite automata NFAs $\hat{\mathcal{A}}^\varphi$, and $\hat{\mathcal{A}}^{\neg\varphi}$ (4), and, then, in the equivalent[11] DFAs $\tilde{\mathcal{A}}^\varphi$ and $\tilde{\mathcal{A}}^{\neg\varphi}$ (5). Finally, the product of $\tilde{\mathcal{A}}^\varphi$ and $\tilde{\mathcal{A}}^{\neg\varphi}$ is

[11]For each NFA we can always find a DFA recognizing the same formal language.

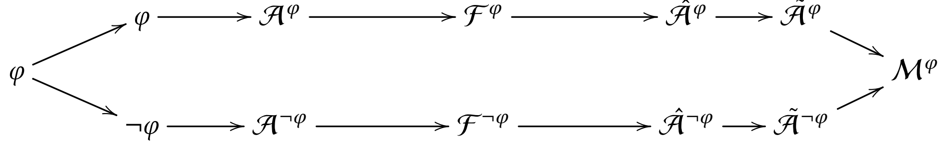

FIGURE 3.3. Steps required to generate an FSM from an LTL formula φ

computed, and from it the final FSM $\mathcal{M}^{\varphi}$ is derived by minimization, and by classifying the states in the following way: (q, q') returns $\top$ iff q' is not final in $\tilde{\mathcal{A}}^{\neg\varphi}$, $\bot$ iff q is not final in $\tilde{\mathcal{A}}^{\varphi}$, and ? if both q and q' are final in $\tilde{\mathcal{A}}^{\varphi}$, and $\tilde{\mathcal{A}}^{\neg\varphi}$, respectively.

3.7 Model Checking Agent Programming Languages

Althought model checking was born more than 35 years ago, its application to MAS is almost recent (Bordini et al., 2004, 2006; Kacprzak, Lomuscio, and Penczek, 2004; Raimondi and Lomuscio, 2007). This is due to the fact that the verification of agent-oriented programs poses new challenges that have not yet been adequately addressed, particularly within the context of practical model checking tools. For instance, in agent-based systems is vital to verify not only the behaviour that the system has, but also why the agents are undertaking certain courses of action within the multiagent system.

One of the most widespread model checkers for MAS is the Model Checking Agent Programming Languages (MCAPL) (Dennis et al., 2012).

MCAPL consists of two components:

- The first is the *Agent Infrastructure Layer* (AIL), which is a set of Java classes designed to act as a toolkit for creating interpreters for BDI Agent Programming Languages, such as Jason (Section 3.3). This toolkit is designed to make the construction of such interpreters quick and easy once an operational semantics is provided.
- The second is *Agent JPF* (AJPF), a version of the Java Pathfinder (JPF) model checker (Havelund, 1999; Havelund and Pressburger, 2000; Visser et al., 2003) which has been extended with a property specification language appropriate for agent programs and some Java interfaces suitable for encapsulating multiagent systems in an efficient fashion.

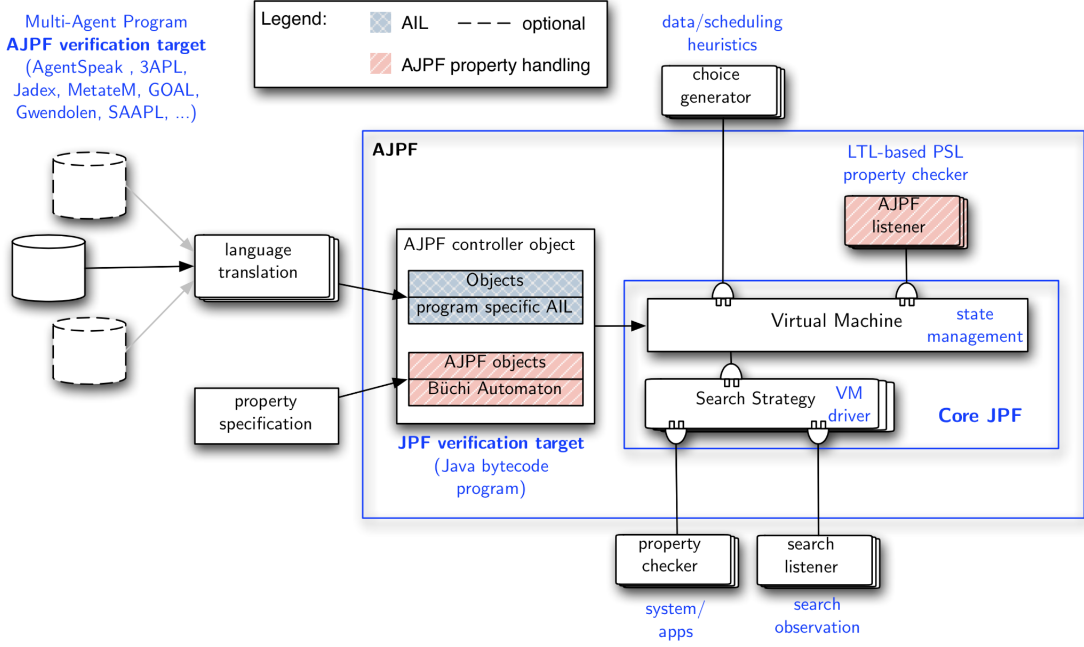

FIGURE 3.4. AJPF architecture (Bordini et al., 2008)

3.8 Hidden Markov Models

A Hidden Markov Model (HMM (Baum and Petrie, 1966; Rabiner and Juang, 1986)) is a statistical Markov model where the system being modeled is assumed to be a Markov process with hidden states. It can be modeled as a quintuple $H = \langle S, A, V, B, \Pi \rangle$ where

- $S = \{s_1, \ldots, s_{N_s}\}$ is the set of states;
- A is the $N_s \times N_s$ transition probability matrix: $A_{i,j} = \Pr(\text{state is } s_j \text{ at time } t+1 \mid \text{state is } s_i \text{ at time } t)$;
- $V = \{v_1, \ldots, v_{N_v}\}$ is the set of observation symbols;
- B is the $N_s \times N_v$ observation probability matrix: $B_{i,j}$, also denoted with $b_i(v_j)$ for clarity, is $\Pr(v_j \text{ is observed at time } t \mid \text{state is } s_i \text{ at time } t)$;
- $\Pi = \{\pi_1, \ldots, \pi_{N_s}\}$ is the initial state distribution: π_i is the probability that the initial state is s_i.

To compute the probability that an HMM H ends in a specific state given an observation sequence $O = \langle O_1, O_2, \ldots, O_T \rangle$, we can use the forward algorithm (Rabiner, 1990). Let $Q = \langle q_1, q_2, \ldots, q_T \rangle$ denote the (unknown) state sequence that the system passed through, i.e., q_t denotes the state of the system when observation O_t is made. Let $\alpha_t(i) = Pr(O_1, O_2, \ldots, O_t, q_t = s_i | H)$, i.e., the probability that the first t observations yield $O_1, O_2, \ldots, O_t$ and that q_t is s_i, given the model H.

The base case is:

$$\alpha_1(j) = \pi_j b_j(O_1) \text{ for } 1 \leq j \leq N_s$$

whereas the recursive case is:

$$\alpha_{t+1}(j) = (\Sigma_{i=1..N_s} \alpha_t(i) A_{i,j}) b_j(O_{t+1}) \text{ for } 1 \leq t \leq T-1 \text{ and } 1 \leq j \leq N_s$$

3.9 Global Types

Global types (Ancona, Drossopoulou, and Mascardi, 2012; Mascardi and Ancona, 2013) are behavioral types for specifying and verifying multiparty protocols involving many distributed components, inspired by the process algebra approach. The constrained global types (Ancona, Barbieri, and Mascardi, 2013) are an extension of the formalism of global types in multiagent systems resulted from a previous work with a mechanism for easily expressing constrained shuffle of message sequences; accordingly, it has been extended the semantics to include the newly introduced feature, and show the expressive power of these "constrained global types". In (Ancona, Barbieri, and Mascardi, 2013) the authors showed how constrained global types can be used to generate monitor agents which were able to check the behaviour of other agents inside the system. Given a protocol expressed using constrained global types, monitors are able to verify the compliance of agents to the protocol.

The two key concepts underlying global types are *events* and *event types*.

EVENTS. An event e is any observable event taking place in the MAS environment, including communicative actions, actions performed by agents, agent location and moves, and actions performed by artifacts. We do not face the transduction problem and assume that events are translated into symbols that agents can manipulate by some mediator between the agents and the environment.

EVENT TYPES. From a logical point of view, an event type ϑ is a predicate on events. Its interpretation is the set of events that verify ϑ; we write $e \in \vartheta$ to mean that ϑ is true on e, and we also say that e has type ϑ.
We can better explain the event types with an example:

```
transport(policeman(marcus), prisoner(alice),
from(jail), to(room1), by(car))
∈ move(alice, jail, room1).
```

With respect to the actual event that took place in the environment and that was transduced into a symbolic form, the event type may be identified by a different functor symbol with different arguments (like in example, where "move(...)" is the event type of a "transport(..)" event) and may abstract some details which are not relevant for the monitoring activities.

3.9.1 Syntax

The protocol specification using global types represents a set of possibly infinite traces of events and is defined on top of the following operators:
• ϵ (*empty trace*), representing the singleton set $\{\epsilon\}$ containing the empty trace ϵ of events.
• $\vartheta^n{:}\tau$ (*sequence with producer*), representing the set of all traces whose first event e matches the event type ϑ ($e \in \vartheta$), and the remaining part is a trace in the set represented by τ. The integer n specifies the least required number of times $e \in \vartheta$ has to be "consumed" to allow a transition labeled by e. Each occurrence of a producer event type must correspond to the occurrence of a new event; in contrast, consumer event types correspond to the same event specified by a certain producer event type. The purpose of consumer event types is to impose constraints between branches of the fork operator, *without introducing new events.*
• $\vartheta{:}\tau$ (*sequence with consumer*), representing the set of all traces where $e \in \vartheta$, and the remaining part is a trace in the set represented by τ. ϑ must match with a producer ϑ^n event type available in another fork branch of the protocol.
• $\tau_1{+}\tau_2$ (*choice*), representing the union of the traces of τ_1 and τ_2.
• $\tau_1|\tau_2$ (*fork*), representing the set obtained by shuffling the traces in τ_1 with the traces in τ_2.
• $\tau_1 \cdot \tau_2$ (*concat*), representing the set of traces obtained by concatenating the traces of τ_1 with those of τ_2.

Global types are regular terms, that is, can be cyclic (recursive), and hence they can be represented by a finite set of syntactic equations. To make the treatment simpler, we limit our investigation to *contractive* (a.k.a. *guarded*) and *deterministic* global types. A global type τ is *contractive* if all infinite paths[12] in τ contain an occurrence of the ":" constructor. Determinism ensures that dynamic checking can be performed efficiently without backtracking. Intuitively, a global type is deterministic if, in case more transition rules can be applied when event e takes place, they lead to equivalent global types.

3.9.2 Semantics

The state of a global type τ can be represented by τ itself. In this section, when talking about global types we will refer to their current state. Also, we will use "global type" and "protocol" interchangeably.

The interpretation of a global type is based on the notion of transition, a total function

$next : Pr{\times}Event \rightarrow \mathcal{P}_{fin}\,(Pr)$,

where Pr and $Event$ denote the set of contractive global types and of events, respectively.

If τ_1 represents the current state of the protocol and the event e takes place, then the protocol can move to τ_2 iff $next(\tau_1, e) = \tau_2$, that we write as $\tau_1 \overset{e}{\rightarrowtail} \tau_2$.

[12] By "path of a global type" we mean "path in the possibly infinite tree corresponding to the term that represents the global type".

The auxiliary function $\epsilon(_)$ specifies the global types whose interpretation contains the empty sequence ϵ. Intuitively, a global type τ s.t. $\epsilon(\tau)$ holds specifies a protocol that is allowed to successfully terminate.

Let τ_0 be a contractive global type. A *run* ρ for τ_0 is a sequence $\tau_0 \xrightarrow{e_0} \tau_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{n-1}} \tau_n \xrightarrow{e_n} \tau_{n+1} \xrightarrow{e_{n+1}} \ldots$ such that (1) either the sequence is infinite, or it has finite length $k \geq 0$ and the last global type τ_k verifies $\epsilon(\tau_k)$; and (2) for all τ_i , e_i , and τ_{i+1} in the sequence, $\tau_i \xrightarrow{e_i} \tau_{i+1}$ holds.

We denote by A(ρ) the possibly empty or infinite sequence of events $e_0 e_1 \ldots e_n \ldots$ contained in ρ. The interpretation $[\![\tau_0]\!]$ of τ_0 is the set $\{A(\rho) | \rho \text{ is a run for } \tau_0\}$. A contractive global type τ is deterministic if for e any possible run ρ of τ and any possible τ' in ρ, if $\tau' \xrightarrow{e} \tau''$, and $\tau' \xrightarrow{e} \tau'''$, then $[\![\tau'']\!] = [\![\tau''']\!]$.

4 *Trace expressions*

"Don't use words too big for the subject.
Don't say infinitely when you mean very;
otherwise you'll have no word left when you
want to talk about something really infinite."
- C.S. Lewis

In this chapter, we present trace expressions as a constrained global types extension (Section 3.9) and we formally compare their expressive power with LTL, a formalism widely adopted in static and runtime verification (Section 3.6). We show that any LTL formula can be translated into a trace expression which is equivalent from the point of view of runtime verification. Since trace expressions are able to express and verify sets of traces that are not context-free, we can derive that in the context of runtime verification our formalism is more expressive than LTL. Trace expressions are a compact and expressive formalism, which can be employed to model complex interaction protocols, and to generate monitors for the Jason and JADE platforms, and can be generalized to support runtime verification of different kinds of properties and systems.

The contents of this chapter are published in
(Ancona, Ferrando, and Mascardi, 2016)

4.1 Introduction

As we mentioned in Section 3.5.2, there are several specification formalisms employed by RV; some of them are well-known formalisms that have been originally introduced for other aims, as regular expressions, context free grammars, and LTL, while others have been expressly devised for RV.

Trace expressions belong to this latter group; they are an evolution of constrained global types (Section 3.9), which have been initially proposed for RV of agent interactions in multiagent systems in 2013.

Trace expressions are an expressive formalism based on a set of operators (including prefixing, concatenation, shuffle, union, and intersection) to denote finite and infinite traces of events. Their semantics is based on a labeled transition system defined by a simple set of rewriting rules which directly drive the behaviour of monitors generated from trace expressions.

In this chapter we introduce in detail the trace expressions formalism and we formally compare it with LTL (Section 3.6), a formalism which is widely used for RV, even though it was initially introduced for model checking.

When used for RV, the expressive power of LTL is reduced, because at runtime only finite traces can be checked, as already anticipated in Section 3.6.5. For instance, the formula Fp (finally p) which states that an event satisfying the predicate p will eventually occur after a finite trace of other occurred events, can only be partially verified at runtime, because no monitor is able to reject an infinite trace of events that do not satisfy p, which, of course, is not a model for Fp.

To provide a formal account for this limitation, a three-valued semantics for LTL, called LTL_3 has been proposed (Bauer, Leucker, and Schallhart, 2009).

A third truth value "?" is introduced to specify that after a finite trace of events has been occurred, the outcome of a monitor can be inconclusive. For instance, if we consider the formula Fp, and the event e which does not satisfy p, then no monitor generated from Fp is able to decide whether Fp is satisfied or not after the trace eee.

In trace expressions this limitation of RV is modeled by the standard semantics: if the semantics of a trace expression τ contains all finite traces e, ee, eee, $\dots$, then it must also contain the infinite trace $e \dots e \dots$ because no monitor generated from τ will be able to reject it. This corresponds to the more formal claim stating that the semantics of any trace expression is a complete metric space of traces, when the standard distance between traces is considered.

As a consequence, when the standard semantics is considered, one can conclude that LTL and trace expressions are not comparable: neither is more expressive than the other. However, since the two formalisms are considered in the context of RV, if the more appropriate three-valued semantics is considered, then trace expressions are strictly more expressive than the LTL: every LTL formula can be encoded into a trace expression with an equivalent three-valued semantics, whereas the opposite property does not hold, since trace expressions are also able to specify context-free and non context-free languages.

4.2 The trace expression formalism

Trace expressions introduce a novelty with respect to constrained global types: besides the union (a.k.a. choice), concatenation, and shuffle (a.k.a. fork) operators, trace expressions support intersection as well. Intersection replaces the constrained shuffle operator (Ancona, Barbieri, and Mascardi, 2013; Ancona et al., 2015a), an extension of the shuffle operator introduced for making constrained global types more expressive. Constrained shuffle imposes synchronization constraints on the events inside a shuffle, thus making constrained global types and their semantics more complex; furthermore, constrained shuffle is not compositional: it cannot be expressed as an operation between sets of event traces (that is, the mathematical entities denoted by trace expressions). In contrast, the intersection operator has a simple, intuitive, and compositional semantics (as suggested by the name itself) and yet is very expressive; for instance, as shown in Section 4.3, it can be used for specifying non context-free sets of event traces.

4.2.1 Events

In the following we denote by $\mathcal{E}$ a fixed universe of events. An event trace over $\mathcal{E}$ is a possibly infinite sequence of events in $\mathcal{E}$. In the rest of the thesis the meta-variables e, w, σ and u will range over the sets $\mathcal{E}$, $\mathcal{E}^\omega$, $\mathcal{E}^*$, and $\mathcal{E}^\omega \cup \mathcal{E}^*$, respectively; juxtaposition $e\,u$ denotes the trace where e is the first event, and u is the rest of the trace. A trace expression over $\mathcal{E}$ denotes a set of event traces over $\mathcal{E}$.

As a possible example, we might have

$$\mathcal{E} = \{o.m \mid o \text{ object identity}, m \text{ method name}\}$$

where the event $o.m$ corresponds to an invocation of method named[1] m on the target object o. This is a typical example of set of events arising when monitoring object-oriented systems (we will show an example later on).

4.2.2 Event types

Like constrained global types, also trace expressions are built on top of event types (chosen from a set $\mathcal{ET}$), rather than of single events; an event type denotes a subset of $\mathcal{E}$, and corresponds to a predicate of arity $k \geq 1$, where the first implicit argument corresponds to the event e under consideration; referring to the example where events are method invocations, we may introduce the type $\mathit{safe}(o)$ of all safe method invocations for a given object o, defined by the predicate safe of arity 2 s.t. $\mathit{safe}(e, o)$ holds iff $e = o.\mathit{isEmpty}$.

The first argument of the predicate is left implicit in the event type, and we write $e \in \mathit{safe}(o)$ to mean that $\mathit{safe}(e, o)$ holds. Similarly, the set of events

[1] Here, for simplicity, an event does not include the signature of the method as it should be the case for those languages supporting static overloading.

specified by an event type ϑ is denoted by $[\![\vartheta]\!]$; for instance, $[\![safe(o)]\!] = \{e \mid e \in safe(o)\}$.

For generality, we leave unspecified the formalism used for defining event types; however, in practice we do not expect that much expressive power is required. For instance, for all examples presented in this work a formalism less powerful than regular expressions is sufficient.

4.2.3 Trace expressions

Similarly to a global type, whose syntax was described in Section 3.9, a trace expression τ represents a set of possibly infinite event traces, and is defined on top of the following operators[2], in which, only the *intersection* operator is the new operator introduced with the trace expressions, all the others are inherited from the global types:

- ϵ (empty trace), denoting the singleton set $\{\epsilon\}$ containing the empty event trace ϵ.
- $\vartheta{:}\tau$ (*prefix*), denoting the set of all traces whose first event e matches the event type ϑ ($e \in \vartheta$), and the remaining part is a trace of τ.
- $\tau_1{\cdot}\tau_2$ (*concatenation*), denoting the set of all traces obtained by concatenating the traces of τ_1 with those of τ_2.
- $\tau_1{\wedge}\tau_2$ (*intersection*), denoting the intersection of the traces of τ_1 and τ_2.
- $\tau_1{\vee}\tau_2$ (*union*), denoting the union of the traces of τ_1 and τ_2.
- $\tau_1|\tau_2$ (*shuffle*), denoting the set obtained by shuffling the traces in τ_1 with the traces in τ_2.

To support recursion without introducing an explicit construct, trace expressions are regular (a.k.a. rational or cyclic) terms, as well as the constrained global types: they correspond to trees where nodes are either the leaf ϵ, or the node (corresponding to the prefix operator) ϑ with one child, or the nodes $\cdot$, $\wedge$, $\vee$, and $|$ all having two children. According to the standard definition of rational trees, their depth is allowed to be infinite, but the number of their subtrees must be finite. As originally proposed by Courcelle (Courcelle, 1983), such regular trees can be modeled as partial functions from $\{0, 1\}^*$ to the set of nodes (in our case $\{\epsilon, \cdot, \wedge, \vee, |\} \cup \mathcal{ET}$) satisfying certain conditions.

A regular term can be represented by a finite set of syntactic equations, as happens, for instance, in most modern Prolog implementations where unification supports cyclic terms.

As an example of non recursive trace expression, let $\mathcal{E}$ be the set $\{e_1, \ldots, e_7\}$, and ϑ_i, $i = 1, \ldots, 7$, be the event types such that $e \in \vartheta_i$ iff $e = e_i$ (that is, $[\![\vartheta_i]\!] = \{e_i\}$); then the trace expression

$$TE_1 = ((\vartheta_1{:}\epsilon|\vartheta_2{:}\epsilon){\vee}(\vartheta_3{:}\epsilon|\vartheta_4{:}\epsilon)){\cdot}(\vartheta_5{:}\vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon)$$

[2] Binary operators associate from left, and are listed in decreasing order of precedence, that is, the first operator has the highest precedence.

denotes the following set of event traces:

$$\left\{ \begin{array}{l} e_1e_2e_5e_6e_7, e_1e_2e_5e_7e_6, e_1e_2e_7e_5e_6, e_2e_1e_5e_6e_7, e_2e_1e_5e_7e_6, e_2e_1e_7e_5e_6, \\ e_3e_4e_5e_6e_7, e_3e_4e_5e_7e_6, e_3e_4e_7e_5e_6, e_4e_3e_5e_6e_7, e_4e_3e_5e_7e_6, e_4e_3e_7e_5e_6 \end{array} \right\}$$

As an example of recursive trace expression, if ϑ_i denotes the same event type defined above for $i = 1, \ldots, 7$, and $[\![\vartheta]\!] = \{e_4, e_5, e_6, e_7\}$, $[\![\vartheta']\!] = \{e_1, e_2, e_6, e_7\}$, and $[\![\vartheta'']\!] = \{e_1, e_2, e_3, e_4\}$, then the trace expression

$$\begin{array}{l} TE_2 = (E|\vartheta_1{:}\vartheta_2{:}\vartheta_3{:}\epsilon) \wedge (E'|\vartheta_3{:}\vartheta_4{:}\vartheta_5{:}\epsilon) \wedge (E''|\vartheta_5{:}\vartheta_6{:}\vartheta_7{:}\epsilon) \\ E = \epsilon \vee \vartheta{:}E \qquad E' = \epsilon \vee \vartheta'{:}E' \qquad E'' = \epsilon \vee \vartheta''{:}E'' \end{array}$$

denotes the set $\{e_1e_2e_3e_4e_5e_6e_7\}$.

Finally, the recursive trace expressions $T_1 = (\epsilon \vee \vartheta_1{:}T_1) \cdot T_2$, $T_2 = (\epsilon \vee \vartheta_2{:}T_2)$ represent the infinite but regular terms $(\epsilon \vee \vartheta_1{:}(\epsilon \vee \vartheta_1{:}\ldots)) \cdot (\epsilon \vee \vartheta_2{:}(\epsilon \vee \vartheta_2{:}\ldots))$ and $(\epsilon \vee (\vartheta_2{:}(\epsilon \vee (\vartheta_2{:}\ldots))))$, respectively.

In the rest of the work we will limit our investigation to *contractive* (a.k.a. *guarded*) trace expressions (as in Section 3.9 in the case of global types). In Section 3.9 we introduced contractivenes in an informal way. The formal definition is the following:

Definition 2. *A trace expression τ is* contractive *if all its infinite paths contain the prefix operator.*

In contractive trace expressions all recursive subexpressions must be guarded by the prefix operator; for instance, the trace expression defined by $T_1 = (\epsilon \vee (\vartheta{:}T_1))$ is contractive: its infinite path contains infinite occurrences of $\vee$, but also of the : operator; conversely, the trace expression $T_2 = \vartheta{:}T_2 \vee T_2$ is not contractive.

Trivially, every trace expression corresponding to a finite tree (that is, a non cyclic term) is contractive.

For all contractive trace expressions, any path from their root must always reach either a ϵ or a : node in a finite number of steps. Since in this work all definitions over trace expressions treat $\vartheta{:}\tau$ as a base case (that is, the definition is not propagated to the subexpression τ), restricting trace expressions to contractive ones has the advantage that most of the definitions and proofs requires induction, rather than coinduction, despite trace expressions can be cyclic. As a consequence, the implementation of trace expressions becomes considerably simpler. For this reason, in the rest of the thesis we will only consider contractive trace expressions.

As in constrained global types, which use the *next* transition function to move from a protocol state to another one, also in trace expressions we have a transition function (corresponding to the trace expressions semantics) which can be specified by the transition relation $\delta \subseteq \mathcal{T} \times \mathcal{E} \times \mathcal{T}$, where $\mathcal{T}$ and $\mathcal{E}$ denote the set of trace expressions and of events, respectively. As it is customary, we write $\tau_1 \xrightarrow{e} \tau_2$ to mean $(\tau_1, e, \tau_2) \in \delta$.

$$next(\tau_0, e) = \{\tau_1, \tau_2, ..., \tau_n\} \iff \forall_{1 \le i \le n}.(\tau_0, e, \tau_i) \in \delta$$

The set generated from $next(\tau_0, e)$ can be infinite. If the trace expression τ_1 specifies the current valid state of the system, then an event e is considered

$$
(\text{prefix})\frac{}{\vartheta{:}\tau \xrightarrow{e} \tau}\ e{\in}\vartheta \qquad (\text{or-l})\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1{\vee}\tau_2 \xrightarrow{e} \tau_1'} \qquad (\text{or-r})\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1{\vee}\tau_2 \xrightarrow{e} \tau_2'}
$$

$$
(\text{and})\frac{\tau_1 \xrightarrow{e} \tau_1' \quad \tau_2 \xrightarrow{e} \tau_2'}{\tau_1{\wedge}\tau_2 \xrightarrow{e} \tau_1'{\wedge}\tau_2'} \quad (\text{shuffle-l})\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1|\tau_2 \xrightarrow{e} \tau_1'|\tau_2} \qquad (\text{shuffle-r})\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1|\tau_2 \xrightarrow{e} \tau_1|\tau_2'}
$$

$$
(\text{cat-l})\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1{\cdot}\tau_2 \xrightarrow{e} \tau_1'{\cdot}\tau_2} \qquad (\text{cat-r})\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1{\cdot}\tau_2 \xrightarrow{e} \tau_2'}\ \varepsilon(\tau_1)
$$

FIGURE 4.1. Operational semantics of trace expressions

$$
(\varepsilon\text{-empty})\frac{}{\varepsilon(\epsilon)} \qquad (\varepsilon\text{-or-l})\frac{\varepsilon(\tau_1)}{\varepsilon(\tau_1{\vee}\tau_2)} \qquad (\varepsilon\text{-or-r})\frac{\varepsilon(\tau_2)}{\varepsilon(\tau_1{\vee}\tau_2)} \qquad (\varepsilon\text{-shuffle})\frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1|\tau_2)}
$$

$$
(\varepsilon\text{-cat})\frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1{\cdot}\tau_2)} \qquad (\varepsilon\text{-and})\frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1{\wedge}\tau_2)}
$$

FIGURE 4.2. Empty trace containment

valid iff there exists a transition $\tau_1 \xrightarrow{e} \tau_2$; in such a case, τ_2 will specify the next valid state of the system after event e. Otherwise, the event e is not considered to be valid in the current state represented by τ_1. Figure 4.1 defines the inductive rules for the transition function.

While the transition relation δ with its corresponding rules in Figure 4.1 defines the non empty traces of a trace expression, the predicate $\varepsilon(_)$, inductively defined by the rules in Figure 4.2, defines the trace expressions that contain the empty trace ϵ. If $\varepsilon(\tau)$ holds, then the empty trace is a valid trace for τ.

Rule (prefix) states that valid traces of $\vartheta{:}\tau$ can only start with an event e of type ϑ (side condition $e \in \vartheta$), and continue with traces in τ.

Rules (or-l) and (or-r) state that the only valid traces of $\tau_1{\vee}\tau_2$ have shape $e\ u$, where either $e\ u$ is valid for τ_1 (rule (or-l)), or $e\ u$ is valid for τ_2 (rule (or-r)).

Rule (and) states that the only valid traces of $\tau_1{\wedge}\tau_2$ have shape $e\ u$, where $e\ u$ is valid for both τ_1 and τ_2.

Rules (shuffle-l) and (shuffle-r) state that the only valid traces of $\tau_1|\tau_2$ have shape $e\ u$, where either $e\ u_1'$ and u_2 are valid traces for τ_1 and τ_2, respectively, and u can be obtained as the shuffle of u_1' with u_2 (rule (shuffle-l)), or u_1 and $e\ u_2'$ are valid traces for τ_1 and τ_2, respectively, and u can be obtained as the shuffle of u_1 with u_2' (rule (shuffle-r)).

Rules (cat-l) and (cat-r) state that the only valid traces of $\tau_1{\cdot}\tau_2$ have shape $e\ u$, where either $e\ u_1'$ and u_2 are valid traces for τ_1 and τ_2, respectively, and u can be obtained as the concatenation of u_1' to u_2 (rule (cat-l)), or ϵ is a valid trace for τ_1 (side condition $\varepsilon(\tau_1)$) and $e\ u$ is a valid trace for τ_2 (rule (cat-r)).

For what concerns Figure 4.2, rules (ε-shuffle), (ε-cat) and (ε-and) require the empty trace to be contained in both subexpressions τ_1 and τ_2, whereas for the union operator it suffices that the empty trace is contained in either τ_1

(rule (ε-or-l)) or τ_2 (rule (ε-or-r)). The prefix operator can never build sets of traces containing the empty trace, whereas ϵ contains just the empty trace (rule (ε-empty)).

The set of traces $[\![\tau]\!]$ denoted by a trace expression τ is defined in terms of the transition relation δ, and the predicate $\varepsilon(_)$. Since $[\![\tau]\!]$ may contain infinite traces, the definition of $[\![\tau]\!]$ is coinductive.

Definition 3. *For all possibly infinite event traces u and trace expressions τ, $u \in [\![\tau]\!]$ is coinductively defined as follows:*

- *either $u = \epsilon$ and $\varepsilon(\tau)$ holds,*
- *or $u = e\,u'$, and there exists τ' s.t. $\tau \xrightarrow{e} \tau'$ and $u' \in [\![\tau']\!]$ hold.*

In the following we will need to consider the reflexive and transitive closure of the transition relation: if σ is a finite (possibly empty) event trace, then the relation $\tau \xrightarrow{\sigma} \tau'$ is inductively defined as follows: $\tau \xrightarrow{\sigma} \tau'$ holds iff

- $\sigma = \epsilon$, and $\tau' = \tau$;
- or $\sigma = e\,\sigma'$, and there exists τ'' s.t. $\tau \xrightarrow{e} \tau''$, and $\tau'' \xrightarrow{\sigma'} \tau'$.

Let us consider again the previous examples of trace expressions:

$$
\begin{array}{l}
TE_1 = ((\vartheta_1{:}\epsilon|\vartheta_2{:}\epsilon)\vee(\vartheta_3{:}\epsilon|\vartheta_4{:}\epsilon))\cdot(\vartheta_5{:}\vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon)\\
TE_2 = (E|\vartheta_1{:}\vartheta_2{:}\vartheta_3{:}\epsilon)\wedge(E'|\vartheta_3{:}\vartheta_4{:}\vartheta_5{:}\epsilon)\wedge(E''|\vartheta_5{:}\vartheta_6{:}\vartheta_7{:}\epsilon)\\
E = \epsilon\vee\vartheta{:}E \qquad E' = \epsilon\vee\vartheta'{:}E' \qquad E'' = \epsilon\vee\vartheta''{:}E''\\
\forall\, i \in \{1..7\}\ [\![\vartheta_i]\!] = \{e_i\} \qquad [\![\vartheta]\!] = \{e_4, e_5, e_6, e_7\}\\
[\![\vartheta']\!] = \{e_1, e_2, e_6, e_7\} \quad [\![\vartheta'']\!] = \{e_1, e_2, e_3, e_4\}
\end{array}
$$

We show that there exist τ_1, τ_2 s.t. $TE_1 \xrightarrow{\sigma_1} \tau_1$, with $\sigma_1 = e_1e_2e_5e_6e_7$, $\varepsilon(\tau_1)$, $TE_2 \xrightarrow{\sigma_2} \tau_2$, with $\sigma_2 = e_1e_2e_3e_4e_5e_6e_7$, and $\varepsilon(\tau_2)$.

For $TE_1 \xrightarrow{\sigma_1} \tau_1$ we have $\vartheta_1{:}\epsilon|\vartheta_2{:}\epsilon \xrightarrow{e_1} \epsilon|\vartheta_2{:}\epsilon \xrightarrow{e_2} \epsilon|\epsilon$, hence $(\vartheta_1{:}\epsilon|\vartheta_2{:}\epsilon)\vee(\vartheta_3{:}\epsilon|\vartheta_4{:}\epsilon) \xrightarrow{e_1e_2} \epsilon|\epsilon$, and $TE_1 \xrightarrow{e_1e_2} (\epsilon|\epsilon)\cdot(\vartheta_5{:}\vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon)$. Furthermore, $\vartheta_5{:}\vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon \xrightarrow{e_5} \vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon \xrightarrow{e_6} \epsilon|\vartheta_7{:}\epsilon \xrightarrow{e_7} \epsilon|\epsilon$, hence $\vartheta_5{:}\vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon \xrightarrow{e_5e_6e_7} \epsilon|\epsilon$, and, because $\varepsilon(\epsilon|\epsilon)$, we can conclude $(\epsilon|\epsilon)\cdot(\vartheta_5{:}\vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon) \xrightarrow{e_5e_6e_7} \epsilon|\epsilon$, hence, $TE_1 \xrightarrow{e_1e_2e_5e_6e_7} \epsilon|\epsilon$.

For $TE_2 \xrightarrow{\sigma_2} \tau_2$ we have $E|\vartheta_1{:}\vartheta_2{:}\vartheta_3{:}\epsilon \xrightarrow{e_1e_2e_3} E|\epsilon \xrightarrow{e_4e_5e_6e_7} E|\epsilon$, and $E'|\vartheta_3{:}\vartheta_4{:}\vartheta_5{:}\epsilon \xrightarrow{e_1e_2} E'|\vartheta_3{:}\vartheta_4{:}\vartheta_5{:}\epsilon \xrightarrow{e_3e_4e_5} E'|\epsilon \xrightarrow{e_6e_7} E'|\epsilon$, and, finally, $E''|\vartheta_5{:}\vartheta_6{:}\vartheta_7{:}\epsilon \xrightarrow{e_1e_2e_3e_4} E''|\vartheta_5{:}\vartheta_6{:}\vartheta_7{:}\epsilon \xrightarrow{e_5e_6e_7} E''|\epsilon$. Therefore $TE_2 \xrightarrow{e_1e_2e_3e_4e_5e_6e_7} (E|\epsilon)\wedge(E'|\epsilon)\wedge(E''|\epsilon)$ and $\varepsilon(E|\epsilon)$, $\varepsilon(E'|\epsilon)$, and $\varepsilon(E''|\epsilon)$, hence $\varepsilon((E|\epsilon)\wedge(E'|\epsilon)\wedge(E''|\epsilon))$.

Since the semantics of trace expressions is coinductive, they can specify non terminating behaviour; for instance, the trace expression defined by $T = \vartheta_1{:}T$ denotes the set with just the infinite trace $e_1\,e_1 \ldots e_1 \ldots$ containing infinite occurrences of e_1; had we considered an inductive semantics, T would have denoted the empty set. For the very same reason, the trace expression defined by $T' = \epsilon\vee\vartheta_1{:}T'$ denotes the set containing all finite traces of the event e_1,

but also the infinite trace $e_1\, e_1 \ldots\; e_1 \;\ldots$. From the point of view of RV, the only difference between the two types is that for T' the monitored system is allowed to halt at any time, whereas for T the system can never stop.

Since at runtime it is not possible to check that a given monitored system will always eventually stop, trace expressions cannot denote sets of traces which are not complete metric spaces, with the standard distance between traces: $d(u_1, u_2) = 2^{-n}$, where n denotes the smallest index (starting from 0) at which the two traces are different; by convention, if the two traces are equal, than $n = \infty$, and $2^{-n} = 0$. For instance, if the semantics of a trace expression τ contains traces of arbitrarily large length of the event e_1, then it also contains the infinite trace $e_1\, e_1 \ldots\; e_1 \;\ldots$; indeed, the monitor associated with τ will not be able to reject it.

Such a limitation is independent of the used formalism, but it is intimately related to RV; as pointed out in Section 4.5, similar issues arise when the LTL is used for RV: its semantics has to be revisited to take into account the fact that at runtime only finite traces can be monitored and checked.

4.2.4 Deterministic trace expressions

As anticipated in in Section 3.9, constrained global types can be either deterministic or nondeterministic. This also applies to trace expressions, indeed, there are trace expressions τ for which the problem of word recognition is less efficient because of non determinism.

In the previous section, we have presented the syntax of trace expressions as a constrained global types syntax evolution highlighting the main differences among the operators; in particular, trace expressions have the same constrained global types operators with in addition the *intersection* operator.

Non determinism originates from the union, shuffle, and concatenation operators, because for each of them two possibly overlapping transition rules are defined; consequently, the new *intersection* operator does not influence the trace expressions *determinism*.

Let us consider the trace expression $\tau = (\vartheta_1{:}\vartheta_2{:}\epsilon)\vee(\vartheta_1{:}\vartheta_3{:}\epsilon)$, where $[\![\vartheta_i]\!] = \{e_i\}$ for $i \in \{1..3\}$. Both transitions $\tau \xrightarrow{e_1} \vartheta_2{:}\epsilon$ and $\tau \xrightarrow{e_1} \vartheta_3{:}\epsilon$ are valid, but $[\![\vartheta_2{:}\epsilon]\!] \neq [\![\vartheta_3{:}\epsilon]\!]$; therefore, to correctly accept the trace $e_1 e_3$, both rules have to be applied simultaneously, and the set of trace expressions $\{\vartheta_2{:}\epsilon, \vartheta_3{:}\epsilon\}$ has to be considered, as it is done for non deterministic automaton.

Similarly, for the trace expression $\tau' = (\vartheta_1{:}\vartheta_2{:}\epsilon)|(\vartheta_1{:}\vartheta_3{:}\epsilon)$, both transitions $\tau' \xrightarrow{e_1} (\vartheta_2{:}\epsilon)|(\vartheta_1{:}\vartheta_3{:}\epsilon)$ and $\tau' \xrightarrow{e_1} (\vartheta_1{:}\vartheta_2{:}\epsilon)|(\vartheta_3{:}\epsilon)$ are valid, but $[\![(\vartheta_2{:}\epsilon)|(\vartheta_1{:}\vartheta_3{:}\epsilon)]\!] \neq$

$[\![(\vartheta_1{:}\vartheta_2{:}\epsilon)|(\vartheta_3{:}\epsilon)]\!]$.

Finally, for the trace expression $\tau'' = (\epsilon\vee\vartheta_1{:}\vartheta_2{:}\epsilon)\cdot(\vartheta_1{:}\epsilon)$ both transitions $\tau'' \xrightarrow{e_1} (\vartheta_2{:}\epsilon)\cdot(\vartheta_1{:}\epsilon)$ and $\tau'' \xrightarrow{e_1} \epsilon$ are valid, but $[\![(\vartheta_2{:}\epsilon)\cdot(\vartheta_1{:}\epsilon)]\!] \neq [\![\epsilon]\!]$.

In the rest of this work we will focus on deterministic trace expressions: indeed, the problem of word recognition is simpler and more efficient in the deterministic case.

Deterministic trace expressions are defined as follows.

Definition 4. *Let τ be a trace expression; τ is* deterministic *if for all finite event traces σ, if $\tau \xrightarrow{\sigma} \tau'$ and $\tau \xrightarrow{\sigma} \tau''$ are valid, then $[\![\tau']\!] = [\![\tau'']\!]$.*

The trace expressions τ, τ', and τ'', as defined above, are not deterministic, while the respectively equivalent trace expressions $\vartheta_1{:}(\vartheta_2{:}\epsilon \vee \vartheta_3{:}\epsilon)$, $\vartheta_1{:}(((\vartheta_2{:}\epsilon)|(\vartheta_1{:}\vartheta_3{:}\epsilon)) \vee ((\vartheta_1{:}\vartheta_2{:}\epsilon)|(\vartheta_3{:}\epsilon)))$, and $\vartheta_1{:}(\epsilon \vee \vartheta_2{:}\vartheta_1{:}\epsilon)$ are deterministic.

4.2.5 Expansive trace expressions

The trace expressions expressivity is due to the presence of *expansive* terms.

Definition 5. *A trace expression τ is expansive iff $\tau = \tau_1 \cdot \tau_2$ and τ_1 is a cyclic term containing τ; or $\tau = \tau_1 | \tau_2$ and either τ_1 or τ_2 is a cyclic term containing τ; or $\tau = \tau_1 \wedge \tau_2$ and either τ_1 or τ_2 is a cyclic term containing τ; or it contains a subtrace that is expansive.*

Expansive subtraces allow the trace expression formalism to recognize more than context-free languages. Given a trace expression τ, $exp(\tau)$ is true if τ is expansive.

Example 1. *An example of an* expansive *concatenation term is:*
$[\![\vartheta_1]\!] = \{e_1\} \quad [\![\vartheta_2]\!] = \{e_2\}$
$\tau = \tau_1 \cdot \tau_2 \quad \tau_1 = (\vartheta_1{:}\tau) \vee \epsilon \quad \tau_2 = \vartheta_2{:}\epsilon$
$\tau \xrightarrow{e_1} (\tau_1 \cdot \tau_2) \cdot \tau_2 \xrightarrow{e_1} ((\tau_1 \cdot \tau_2) \cdot \tau_2) \cdot \tau_2 \xrightarrow{e_1} (((\tau_1 \cdot \tau_2) \cdot \tau_2) \cdot \tau_2) \cdot \tau_2 \xrightarrow{e_1} \ldots$

Example 2. *An example of an* expansive *shuffle term is:*
$[\![\vartheta_1]\!] = \{e_1\} \quad [\![\vartheta_2]\!] = \{e_2\}$
$\tau = \tau_1 | \tau_2 \quad \tau_1 = \vartheta_1{:}\epsilon \quad \tau_2 = \vartheta_2{:}\tau$
$\tau \xrightarrow{e_2} \tau_1|(\tau_1|\tau_2) \xrightarrow{e_2} \tau_1|(\tau_1|(\tau_1|\tau_2)) \xrightarrow{e_2} \tau_1|(\tau_1|(\tau_1|(\tau_1|\tau_2))) \xrightarrow{e_2} \ldots$

Non-expansive trace expressions are defined as follows.

Definition 6. *Let τ be a trace expression; τ is* non-expansive *if it does not contain neither expansive concatenations nor expansive shuffles terms.*

4.2.6 Derived operators

We first introduce some useful operators that will be used in the rest of the thesis.

CONSTANTS. The constants 1 and 0 denote the set of all possible traces over $\mathcal{E}$ and the empty set, respectively. Constant 1 is equivalent to the expression $T = \epsilon \vee any{:}T$, where any is the event type s.t. $[\![any]\!] = \mathcal{E}$; constant 0 is equivalent to the expression $none{:}\epsilon$, where $none$ is the event type s.t. $[\![none]\!] = \emptyset$.

FILTER OPERATOR. The filter operator is useful for making trace expressions more compact and readable. The expression $\vartheta \gg \tau$ denotes the set of all traces contained in τ, when deprived of all events that do not match ϑ. Assuming that event types are closed by complementation, the expression above is a convenient syntactic shortcut for $T|\tau$, where $T = \epsilon \vee \overline{\vartheta}{:}T$, and $\overline{\vartheta}$ is the complement event type of ϑ, that is, $[\![\overline{\vartheta}]\!] = \mathcal{E} \setminus [\![\vartheta]\!]$.

The corresponding rules for the transition relation and the auxiliary function $\varepsilon(_)$ can be easily derived:

$$\text{(cond-t)}\frac{\tau \xrightarrow{e} \tau'}{\vartheta \gg \tau \xrightarrow{e} \vartheta \gg \tau'}\ e \in \vartheta \qquad \text{(cond-f)}\frac{}{\vartheta \gg \tau \xrightarrow{e} \vartheta \gg \tau}\ e \notin \vartheta \qquad (\varepsilon\text{-cond})\frac{\varepsilon(\tau)}{\varepsilon(\vartheta \gg \tau)}$$

4.3 *Examples of specifications with trace expressions*

In this section we provide some examples to show the expressive power of trace expressions. Unless specified otherwise, in the rest of the thesis we will consider singleton event types, that are, event types representing a single event; with abuse of notation, we will abbreviate events with their corresponding singleton event types.

4.3.1 Ping Pong Protocol

We start the examples with a simple ping-pong protocol.

Let *ping* and *pong* denote the event types s.t.

$$[\![ping]\!] = \{send(alice, bob, _)\}$$

$$[\![pong]\!] = \{send(bob, alice, _)\}$$

The following trace expression specifies the protocol where at each step alice sends a message to bob, and then bob replies to it, and the number of steps is arbitrary (even infinite) but greater than 0.

$$PingPong = ping{:}pong{:}Forever$$
$$Forever = \epsilon \vee ping{:}pong{:}Forever$$

The protocol specified by *PingPong* is allowed to terminate after one or more steps; the following variation specifies a ping-pong protocol which is not allowed to terminate:

$$PingPongForever = ping{:}pong{:}PingPongForever$$

In terms of monitoring, the only difference between *PingPong* and *PingPongForever* consists in the fact that in the latter case an anomaly of the system is reported if no event is detected after a predefined timeout has expired.

4.3.1.1 Stack objects

We expand the example[3] where events correspond to method invocations on objects; besides the already introduced event type $safe(o)$ s.t. $e \in safe(o)$ iff $e = o.isEmpty$, we define the following other event types:

$$[\![pop(o)]\!] = \{o.pop\},\ [\![top(o)]\!] = \{o.top\},\ [\![push(o)]\!] = \{o.push\};$$

$$[\![stack(o)]\!] = \{o.pop, o.top, o.push, o.isEmpty\};$$

$$[\![unsafe(o)]\!] = \{o.pop, o.top, o.push\}.$$

Our purpose is to specify through a trace expression *Stack* all safe traces of method invocations on a stack object o which we assume to be initially empty. Safety requires that methods *top* and *pop* can never be invoked on o when o represents the empty stack.

More in details, a trace of method invocations on a given object having identity o is correct iff any finite prefix does not contain more $pop(o)$ event types than $push(o)$, and the event type $top(o)$ can appear only if the number of $pop(o)$ event types is strictly less than the number of $push(o)$ event types occurring before $top(o)$.

The trace expression *Stack* is defined as follows:

$$\begin{array}{rcl} Stack & = & Any \wedge unsafe(o) \gg Unsafe \\ Unsafe & = & \epsilon \vee (push(o){:}(Unsafe | (Tops \cdot (pop(o){:}\epsilon \vee \epsilon)))) \\ Any & = & \epsilon \vee stack(o){:}Any \\ Tops & = & \epsilon \vee top(o){:}Tops \end{array}$$

A correct stack trace is specified by *Stack* which is the intersection of *Any* and $unsafe(o) \gg Unsafe$; *Any* specifies any possible trace of method invocations on stack objects, whereas if an event has type $unsafe(o)$, then it has to verify the trace expression *Unsafe*, which requires that a *push* event must precede a possible empty trace of *top* events, which, in turn, must precede an optional event *pop*; the expression is recursively shuffled with itself, since any *push* event can be safely shuffled with a *top* or a *pop* event.

The specification is deterministic.

To make an example, we can consider $Stack \xrightarrow{\sigma} \tau$ with

$$\sigma = push(o)\, push(o)$$

$$\tau = Any \wedge unsafe(o) \gg (Unsafe | Tops \cdot ((pop(o){:}\epsilon) \vee \epsilon) | Tops \cdot ((pop(o){:}\epsilon) \vee \epsilon))$$

We may observe that $\tau \xrightarrow{e} \tau_1$ and $\tau \xrightarrow{e} \tau_2$, with[4]

$$\begin{array}{rcl} e & = & pop(o) \\ \tau_1 & = & Any \wedge unsafe(o) \gg (Unsafe | \epsilon | Tops \cdot ((pop(o){:}\epsilon) \vee \epsilon)) \\ \tau_2 & = & Any \wedge unsafe(o) \gg (Unsafe | Tops \cdot ((pop(o){:}\epsilon) \vee \epsilon) | \epsilon) \end{array}$$

but $[\![\tau_1]\!] = [\![\tau_2]\!]$.

[3] The example we considered at the beginning of this chapter.

[4] For efficiency reasons, our implementation exploits simplification opportunities after each transition step, therefore in practice for this example the two transitions would lead to the same expression.

4.3.2 Alternating Bit Protocol

A more complex example concerning interactions is the alternating bit protocol (ABP), as defined by Deniélou and Yoshida (Deniélou and Yoshida, 2012), where two parties, Alice and Bob, are involved, and four different types of events can occur: Alice sends a first kind of message to Bob (event type msg_1), Alice sends a second kind of message to Bob (event type msg_2), Bob replies to Alice with an acknowledge to the first kind of message (event type ack_1), Bob replies to Alice with an acknowledge to the second kind of message (event type ack_2). The protocol has to satisfy the following constraints for all event occurrences:

- The n-th occurrence of the event of type msg_1 must precede the n-th occurrence of the event of type msg_2, which, in turn, must precede the $(n+1)$-th occurrence of the event of type msg_1.
- The n-th occurrence of the event of type msg_1 must precede the n-th occurrence of the event of type ack_1, which, in turn, must precede the $(n+1)$-th occurrence of the event of type msg_1.
- The n-th occurrence of the event of type msg_2 must precede the n-th occurrence of the event of type ack_2, which, in turn, must precede the $(n+1)$-th occurrence of the event of type msg_2.

The protocol can be specified by the following trace expression (starting from variable $AltBit_1$):

$$\begin{array}{rclrcl} AltBit_1 &=& msg_1{:}M_2 & AltBit_2 &=& msg_2 : M_1 \\ M_1 &=& msg_1{:}A_2 \vee ack_2{:}AltBit_1 & M_2 &=& msg_2{:}A_1 \vee ack_1{:}AltBit_2 \\ A_1 &=& ack_1{:}M_1 \vee ack_2{:}ack_1{:}AltBit_1 & A_2 &=& ack_2{:}M_2 \vee ack_1{:}ack_2{:}AltBit_2 \end{array}$$

In this case the prefix and union operators are sufficient for specifying the correct behaviour of the system, however, the corresponding trace expression is not very readable. More importantly, if only the prefix and union operators are employed, the size of the expressions grows exponentially with the number of different involved event types.

This problem can be avoided by the use of the intersection and filter operators.

Let $msg_ack(i)$, $i \in \{1..2\}$, and msg denote the event types s.t. $[\![msg_ack(i)]\!] = [\![msg_i]\!] \cup [\![ack_i]\!]$, $i \in \{1..2\}$, and $[\![msg]\!] = [\![msg_1]\!] \cup [\![msg_2]\!]$. Then the ABP can be specified by the following deterministic trace expression:

$$\begin{array}{rcl} AltBit &=& (msg \gg MM) \wedge (msg_ack(1) \gg MA_1) \wedge (msg_ack(2) \gg MA_2) \\ MM &=& msg_1{:}msg_2{:}MM \\ MA_1 &=& msg_1{:}ack_1{:}MA_1 \\ MA_2 &=& msg_2{:}ack_2{:}MA_2 \end{array}$$

The three trace expressions defined by MM, MA_1, and MA_2 correspond to the three constraints informally stated above. The main trace expression $AltBit$ can be easily read as follows: if an event has type msg_1 or msg_2, then it must verify MM, and if an event has type msg_1 or ack_1, then it must verify MA_1, and if an event has type msg_2 or ack_2, then it must verify MA_2.

The trace expression can be easily generalized to k different kinds of messages (with $k \geq 2$), with the size of the expression growing linearly with the number of different involved event types. For instance, for $k = 3$ we have the following trace expression:

$$\begin{array}{rcl} AltBit & = & (msg \gg MM) \wedge (msg_ack(1) \gg MA_1) \wedge \\ & & (msg_ack(2) \gg MA_2) \wedge (msg_ack(3) \gg MA_3) \end{array}$$

$$\begin{array}{rclrcl} MM & = & msg_1{:}msg_2{:}msg_3{:}MM & MA_1 & = & msg_1{:}ack_1{:}MA_1 \\ MA_2 & = & msg_2{:}ack_2{:}MA_2 & MA_3 & = & msg_3{:}ack_3{:}MA_2 \end{array}$$

4.3.3 Non context free languages

Trace expressions allow the specification of non context free languages; let us consider for instance the typical example of non context free language $\{a^n b^n c^n \mid n \geq 0\}$. This language can be specified by the following trace expression (defined by T)

$$T = (a_or_b \gg AB) \wedge (b_or_c \gg BC)$$

$$AB = \epsilon \vee (a{:}(AB \cdot (b{:}\epsilon)))$$

$$BC = \epsilon \vee (b{:}(BC \cdot (c{:}\epsilon)))$$

where $[\![a]\!] = \{a\}$, $[\![b]\!] = \{b\}$, $[\![c]\!] = \{c\}$, $[\![a_or_b]\!] = \{a, b\}$, and $[\![b_or_c]\!] = \{b, c\}$.

Assuming the universe of events $\mathcal{E} = \{a, b, c\}$, the expression $a_or_b \gg AB$ denotes all traces of events over $\mathcal{E}$ that, when restricted to finite length[5] and to events a or b, correspond to the sequence $a^n b^n$ for some $n \in \mathbb{N}$; similarly, the expression $b_or_c \gg BC$ denotes all traces of events over $\mathcal{E}$ that, when restricted to finite length and to events b or c, correspond to the sequence $b^n c^n$ for some $n \in \mathbb{N}$. Therefore the finite traces of expression T, which is the intersection of $a_or_b \gg AB$ and $b_or_c \gg BC$, are the non-context free language $\{a^n b^n c^n \mid n \geq 0\}$.

Although T is deterministic, it has the drawback that non correct traces can be detected with a certain latency. For instance the transition $T \overset{aabc}{\longrightarrow} T'$ holds, with $T' = (a_or_b \gg (b{:}\epsilon)) \wedge (b_or_c \gg \epsilon)$, and clearly $aabc$ is not a valid prefix for the language; however, $[\![T']\!] = \emptyset$, and T' is not able to accept any further event, that is, recognition fails, independently from the next event.

To avoid this problem, the following equivalent (assuming that $\mathcal{E} = \{a, b, c\}$) deterministic trace expression can be employed:

$$\begin{array}{rclrcl} T_2 & = & (AB \cdot C) \wedge (b_or_c \gg BC) & AB & = & \epsilon \vee (a{:}(AB \cdot (b{:}\epsilon))) \\ BC & = & \epsilon \vee (b{:}(BC \cdot (c{:}\epsilon))) & C & = & \epsilon \vee c{:}C \end{array}$$

In this case, $AB \cdot C$ forces events of type c to occur only after all required events of type b have been already occurred. In this case there is no T_2'' s.t. $T_2 \overset{aabc}{\longrightarrow} T_2''$

[5] Recall that for a comparison with context-free languages we need to disregard infinite traces; for instance, $a_or_b \gg AB$ and $b_or_c \gg BC$ contain also the infinite traces a^ω and b^ω, respectively.

holds; indeed, $T_2 \xrightarrow{aab} T_2'$ with

$$T_2' = ((b{:}\epsilon){\cdot}(\epsilon{\vee}(c{:}C))){\wedge}(b_or_c{\gg}(BC{\cdot}(c{:}\epsilon))),$$

and there exists no T_2'' s.t. $T_2' \xrightarrow{c} T_2''$, since the only possible transition from T_2' is $T_2' \xrightarrow{b} T_2''$, with

$$T_2'' = (\epsilon{\vee}(c{:}C)){\wedge}(b_or_c{\gg}((\epsilon{\vee}(b{:}BC{\cdot}(c{:}\epsilon))){\cdot}((c{:}\epsilon){\cdot}(c{:}\epsilon)))),$$

and $[\![T_2'']\!] = \{cc\}$.

4.4 *Trace expressions monitoring*

One possible way to achieve the RV of a system is using one (or more) monitor(s) generated from a formal specification (that is the property we want to verify, for instance a LTL property (Pnueli, 1977), or in our case a trace expression).

Monitoring can be classified with respect to three main aspects:

1. When the monitoring is executed.
    - *online*, the monitor checks the executions incrementally at runtime;
    - *offline*, the monitor checks recorded executions (for instance log files).
2. How the monitoring is implemented (Falcone, 2010).
    - *inline*, the monitor is inserted within the monitored program;
    - *outline*, the monitor runs in a thread or process different from the monitored program.
3. Which kind of errors the monitor arises (d'Amorim and Rosu, 2005).
    - *precise*, the monitor has observed an error in the execution trace analyzed;
    - *predictive*, the monitor indicates errors that have not occurred in the observed execution trace but could possibly occur in other executions of the program.

A possible way to specify the monitor behavior is through the set of all correct traces (finite or infinite sequences of events) which can be generated during the system execution[6]. This set of traces can be defined using different formalisms. In our work we adopt trace expressions (Figure 4.3).

As it will be clearer in the rest of the thesis, with respect to the three main aspects reported above, our RV approach can be classified as: *online*/*offline*[7], *outline*, *precise*.

[6]For instance, the system might be a multiagent system and the events of interest might be messages exchanged among agents which must respect some interaction protocol property, or an object oriented systems, where events subject to monitoring might be method calls.

[7]We can analyze both the execution traces at runtime and the recorded traces (log files).

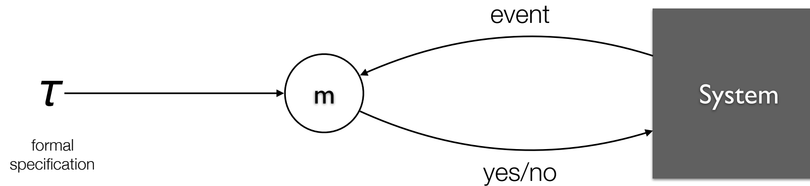

FIGURE 4.3. An abstract view of how to build a monitor.

In Figure 4.3, we reported an abstraction of how we generate the monitors starting from a formal specification, in particular a trace expression in our case.

Once we have defined the property we want to verify using the trace expression formalism, we can automatically build a monitor as a very naive entity – a sniffer – which, each time it observes an event, queries our trace expression in order to check if it is a valid one. Obviously, in order to query the trace expression, the monitor must be able to observe the events generated by the verified system. This task can be more or less easy to achieve, depending on which kind of system we are interested in verifying. Generally, we suppose a way to connect the system and the monitor already exists, or can be developed. In the context of MAS, for instance, if we use JADE, we have already available one – or more – sniffer agents able to intercept the events generated by the agents communication. In such scenario, the implementation of a monitor is extremely easy because it is enough to extend the already existing sniffer. If, for instance, we are interested in verifying a system which does not support any kind of events observer by default, we should be forced to develop it from scratch.

The trace expression semantics has been implemented inside SWI-Prolog[8], a comprehensive free Prolog environment. SWI-Prolog has been chosen particularly because it supports the connection to the most famous and used programming languages (such as Java, making the integration in Jason and JADE very easy), and because it supports coinduction[9] (Milner and Tofte, 1991) through the predefined *coinduction* library[10] (extremely useful since trace expression can be cyclic terms). In Chapter 14 we show how we have implemented the runtime verification process inside the JADE platform. Anyway, once a bridge between the monitor and the system exists, the runtime verification process can be summarized as follows.

Following the trace expression semantics (Section 4.2.3), we use the δ transition function for going from the current state – the trace expression – to the next one using the observed event. If, the event generated by the system is not a valid one, given the trace expression representing the current state, we cannot generate a new state – a new trace expression – using such event. When that

[8] `http://swi-prolog.org/`

[9] For further historical details see Section 4.3 of Sangiorgi's paper (Sangiorgi, 2009).

[10] `http://www.swi-prolog.org/pldoc/doc/_SWI_/library/coinduction.pl`

happens, we have found an error – or better, an unexpected behaviour – and what will be done after to handle it is totally dependent on the specific domain (like stopping the execution of the system or trying to solve the problem).

4.5 Comparison with LTL

In this section we formally prove that trace expressions are more expressive than the LTL, when both formalisms are used for RV. To this purpose we consider the LTL_3 semantics, an adaptation of the standard semantics of LTL formulas expressly introduced to take into account the limitations of RV due to its inability to check infinite traces. Despite there are LTL formulas which do not have an equivalent trace expression according to the standard LTL semantics, when LTL_3 is considered such a difference is no longer exhibited: for any LTL formula φ it is possible to build a contractive and deterministic trace expression τ such that the monitors generated by φ and τ, respectively, are behaviorally equivalent.

4.5.1 Comparing trace expressions and LTL

We have shown in Section 3.6 that LTL formulas as $p\,U q$ cannot be fully verified at runtime, therefore a three-valued semantics LTL_3 has been introduced. To be able to compare LTL formulas with trace expressions, the same three-valued semantics is considered for trace expressions as well.

Given a finite trace $\sigma \in \Sigma^*$ of length $|\sigma| = n$, a continuation of σ is an finite or infinite trace $u \in \Sigma^* \cup \Sigma^\omega$ s.t. for all $0 \leq i < n$ $u(i) = \sigma(i)$.

The three-valued semantics of a trace expression τ is defined as follows:

$$\sigma \in [\![\tau]\!]_3 = \begin{cases} \top & \text{iff } u \in [\![\tau]\!] \text{ for all continuations } u \text{ of } \sigma \\ \bot & \text{iff } u \notin [\![\tau]\!] \text{ for all continuations } u \text{ of } \sigma \\ ? & \text{iff neither of the two conditions above holds} \end{cases}$$

Let us consider again the formula $\varphi = p\,U q$; if we assume that each atomic predicate in AP has a corresponding event type denoted in the same way, then the closest trace expression τ into which φ can be translated is defined by $T = p{:}T \vee q{:}1$, where 1 is the derivable constant introduced in Section 4.3 denoting all possible traces. If we consider the standard semantics we have that, since $\{p\}$ is an event that satisfies p, $\{p\}^\omega \in [\![\tau]\!]$, but $\{p\}^\omega \nvDash \varphi$. However, when considering the three-valued semantics we have that for all $v \in \{\top, \bot, ?\}$ and $\sigma \in \Sigma^*$, $\sigma \vDash \varphi = v$ iff $\sigma \in [\![\tau]\!]_3 = v$. In particular, for all $n \geq 0$, $\{p\}^n \vDash \varphi =?$ and $\{p\}^n \in [\![\tau]\!]_3 =?$.

To translate an LTL formula φ into a trace expression τ s.t. the three-valued semantics is preserved, we exploit the result presented in Section 3. First, φ is translated into an equivalent FSM $\mathcal{M}^\varphi$, then $\mathcal{M}^\varphi$ is translated into an equivalent contractive and deterministic trace expression τ^φ. The latter translation is defined as follows:

- if the initial state returns $\top$, then φ is a tautology, and the corresponding trace expression is the constant 1;

- if the initial state returns $\bot$, then φ is a unsatisfiable, and the corresponding trace expression is the constant 0;
- if the initial state returns ?, then the corresponding trace expression is defined by a finite set of equations $X_1 = \tau_1, \dots, X_n = \tau_n$, where n is the number of states in $\mathcal{M}^\varphi$ that return ?, each of such states is associated with a distinct variable X_i, X_1 is the variable associated with the initial state which corresponds to the whole trace expression τ^φ.

  The expressions τ_i are defined as follows: let k be the number of states $q_1, \dots, q_k$ that do not return $\bot$ for which there exists an incoming edge, labeled with the element $a_i \in 2^{AP}$, from the node associated with X_i; we know that $k > 0$, because the node associated with X_i returns ?. Then $\tau_i = a_1{:}f(q_1) \vee \dots \vee a_k{:}f(q_k)$, where $f(q)$ is defined as follows: if q returns $\top$, then $f(q) = 1$, otherwise (that is, q returns ?), $f(q) = X_q$ (that is, the variable uniquely associated with q is returned).

Since all variables in the expressions $\tau_1, \dots, \tau_n$ are guarded by the prefix operator, τ^φ is contractive; furthermore, it is deterministic because $\mathcal{M}^\varphi$ is deterministic.

Theorem 1. *Let $\mathcal{M}^\varphi$ be the FSM equivalent to φ generated by the procedure described in Chapter 3. Then, the trace expression τ^φ generated from $\mathcal{M}^\varphi$ as specified in this section preserves the semantics of $\mathcal{M}^\varphi$: for all $\sigma \in \Sigma^*$ $\mathcal{M}^\varphi$ accepts σ with output $v \in \{\top, \bot, ?\}$ iff $\sigma \in [\![\tau^\varphi]\!]_3 = v$.*

Proof. The proof proceeds by induction on the length of σ.
Base case: $\sigma = \epsilon$.
The cases where the initial state of the FSM returns $\top$ or $\bot$ are immediate to be proved (in the case of $\bot$ the monitoring ends). The proof when the initial state returns ? is based on the fact that by construction $[\![\tau^\varphi]\!] \neq \emptyset$ and there always exists a trace u s.t. $u \notin [\![\tau^\varphi]\!]$, therefore $\epsilon \in [\![\tau^\varphi]\!]_3 =?$. □
Inductive step: $\sigma = \sigma' e$.
Inductive hypothesis: $\mathcal{M}^\varphi$ accepts σ' with output $v \in \{\top, \bot, ?\}$ and $\sigma' \in [\![\tau^\varphi]\!]_3 = v$.
Without loss of generality, we consider that the monitoring ends immediately when an unexpected event occurs; that is when the FSM visits a $\bot$ node and the trace expression can not move to another state consuming the occurred event. Consequently, $\mathcal{M}^\varphi$ accepts σ' with output $v \in \{\top, ?\}$ and $\sigma' \in [\![\tau^\varphi]\!]_3 = v$.
Analyzing all the possible cases:

- if $v = \top$, for the *inductive hypothesis*, the current state in FSM returns $\top$ and the corresponding trace expression is the constant 1 (by construction); consuming the event e both remain in a state which accepts all incoming events.
- if $v =?$, for the *inductive hypothesis*, the current state in the FSM returns ? and the corresponding trace expression is an equation of the form $X = \tau$ (by construction), which is built as described above (in the previous page). Consequently,

- if the event e brings the FSM in the state $\top$, the trace expression moves in the constant 1 by construction. Both return $\top$;
- if the event e brings the FSM in the state $\bot$, the trace expression can not move to another state by construction and the monitoring ends returning $\bot$. Both return $\bot$;
- if the event e brings the FSM in a state ?, the trace expression moves in a new state represented by an equation of the form $X = \tau$ which is built as described above. Both return ?.

□

Corollary 1. *Trace expressions are strictly more expressive than LTL.*

From (Cohen, Perrin, and Pin, 1993) we know that LTL recognize star-free ω-regular languages, and from Theorem 1 we also know that given a LTL formula φ, we can generate an equivalent trace expression τ^{φ}. Thus, trace expressions are at least expressive as LTL. In Section 4.3.3 we have shown a trace expression τ that specifies a non context free language of traces (when only finite traces are considered). More formally, $\sigma \in [\![\tau \cdot 1]\!]_3 = \top$ iff $\sigma \in \{a^n b^n c^n \mid n \geq 0\}$. This means that for RV (that is, when the three-values semantics is considered) trace expressions are strictly more expressive than LTL, since the LTL is less expressive than ω-regular languages.

5 State of the art

"Every saint has a past, and every sinner has a future."
- Oscar Wilde

In this chapter, we present the formalisms and approaches that are closest to ours. Since we are interested in verifying multiagent systems at runtime, we focused our attention on a state of the art concerning formalisms and approaches used in this context. In particular, we propose and discuss a range of aspects that we consider crucial, and we present and classify the state of the art following them. We leave the comparison with our formalism at the end of the thesis in Chapter 16.

The contents of this chapter are published in (Ancona, Ferrando, and Mascardi, 2018b)

5.1 Engineering Multiagent Systems

When MAS are exploited in real world scenarios, it is extremely common to have to tackle problems correlated to the heterogeneity of agents. We may think about self-driving vehicles, Internet of Things (IoT), Remote Patient Monitoring (RPM), and so on. When the integration of sensors in the MAS comes into play as in these scenarios, the MAS can be seen as a sophisticated cyber-physical system whose dependability must be attained in order to avoid severe failures.

According to (Avizienis et al., 2004), there are many different means to attain dependability: fault prevention, fault tolerance, fault removal, and fault forecasting. Fault prevention is part of general engineering: prevention of development faults is an obvious aim for development methodologies, both for software (e.g., information hiding, modularization, use of strongly-typed programming languages) and hardware (e.g., design rules). Fault tolerance (Avizienis, 1967) is aimed at failure avoidance, and is carried out via error detection and system recovery. Fault removal can take place during development via verification, diagnosis, correction, and during use. Finally, fault forecasting is conducted by performing an evaluation of the system behavior with respect to fault occurrence or activation. Self-adaptation can be seen as a means for a system to achieve fault removal during use. It can be also a means to attain fault tolerance, by detecting the fault source and autonomously adopting strategies that allow the system to be recovered.

In (Weyns, 2018), a self-adaptive system is defined as a system that can handle changes and uncertainties in its environment, the system itself and its goals autonomously (external principle), and that comprises two distinct parts: the first which interacts with the environment and is responsible for the domain concerns; the second which interacts with the first part and monitors its environment, and is responsible for the adaptation concerns (internal principle). The aim of self-adaptation is to let the system collect additional data about the uncertainties during operation, use them to resolve uncertainties, reason about itself, and based on its goals reconfigure or adjust itself to satisfy the changing conditions.

Adaptation may take place in very different ways ranging from switching from the current behavioral protocol to a higher-priority one in systems where protocols are pre-compiled first class entities (Ancona et al., 2015a), to generating behaviour alternatives on-the-fly via an adaptation planner based on a genetic algorithm (Chen et al., 2017), from monitoring a set of target goals of a goal-driven agent looking for the activation of some adaptation trigger (Dalpiaz et al., 2010), to exchanging immediate and retrospective experiences in order to generate new adaptation policies (Jiao and Sun, 2016).

MAS are recognized as an appealing approach for developing decentralized self-adaptive systems (Weyns and Georgeff, 2010); self-adaptive MAS engineered via some IDE, able to undergo static, a priori verification via formal methods, simulation or testing, or to be verified at runtime via monitoring, are suitable to address some of the challenges raised by dependability, besides

those associated with flexibility.

In this chapter we analyze and compare four approaches for engineering complex systems which are based on an explicit and formal model of either the system architecture, or its behavior, or the interactions taking place therein.

The existence of an IDE, the support to parametricity and probability at language level, and the ability to check protocol enactability and to generate protocol-driven code are the dimensions we take into account for stating how much an approach attains **fault prevention**.

Testing, static and dynamic verification abilities, and support to some form of self-adaptation, are the dimensions we take into account for stating how much an approach attains **fault tolerance and fault removal**. Given that the term self-adaptation is not precisely defined in the literature (Weyns, 2018), we leave this dimension vague enough to cover all the possible definitions.

To reduce subjectivity in the choice of the approaches to analyze, we performed a thorough search with both Google Scholar, `scholar.google.com`, and Scopus, `www.scopus.com` and we identified the most promising approaches for engineering self-adaptable MAS taking the number of publications dealing with the approach, the cumulative number of citations (normalized on number of years the approach has been around), and the publications quality into account. We limited our investigation to approaches based on notations amenable for formal reasoning and verification. As an example, despite its undoubted impact on the agent-oriented software engineering development, we do not discuss AUML (Bauer, Müller, and Odell, 2001) because it is not a formal notation.

In our analysis, we took the following ten features into account:

– **Modelled issues**: which aspects of the MAS are modeled in an explicit, automatically processable way?

– **Modeling approach**: which notation/language has been adopted for modeling the aspects above?

– **Integrated development environment (attains fault prevention)**: does the notation/language come with an IDE facilitating its usage?

– **Parametricity (attains fault prevention)**: does the language/notation support some form of parametricity, boosting modularity and reuse of "behavior templates"?

– **Probability (attains fault prevention)**: does the language/notation support some form of probability, boosting robustness and reliability in real systems?

– **Testing/simulation (attains fault tolerance and removal)**: does the approach support testing or simulation of the modelled features?

– **A priori verification (attains fault tolerance and removal)**: does the approach support a priori formal verification?

– **Runtime verification (attains fault tolerance and removal)**: does the approach support runtime verification?

– **Self-adaptation (attains flexibility and fault tolerance and removal)**: does the approach support facilities for engineering self-adaptable MAS, such

as environment monitoring and autonomous behaviour switch based on sensory input?
– **Protocol enactability and protocol-driven code generation (attains fault prevention)**: in case the modelled feature is an agent interaction protocol, does the approach support methods to ensure its enactability? Does it offer tools for automatically generating code for agents that follow the given protocol? If yes, in which agent programming language?
– **Case studies and applications**: Are there case studies showing how to use the notation/language? Has the notation/language been adopted for developing real applications outside academia, targeted to users different from the authors themselves?

The four approaches are presented in alphabetical order. All the descriptions have been validated by the main authors of the approach, acknowledged in the respective sections. At the end of the thesis (in Section 16.1) we evaluate also trace expressions with respect to these features. In Section 16.1.1, we summarize and compare trace expressions with the four following approaches.

5.2 Blindingly Simple Protocol Language

This section has been validated by Amit Chopra and Munindar P. Singh.

BSPL, the blindingly simple protocol language (Singh, 2011a), is a declarative approach based on two main constructs: defining a message schema and composing existing protocols. BSPL treats interaction as first class and puts no constraints on the ordering or occurrence of messages. W.r.t. other approaches to protocol modeling, BSPL relaxes the assumption about point-to-point message ordering as illustrated in the papers mentioned below, and in (Chopra and Singh, 2015b; King et al., 2017).

– **Modelled issues**: interaction protocols from a global perspective.
– **Modeling approach**: a protocol defines a scope within which its roles, parameters, and messages are uniquely named. The roles and parameters of a protocol identify its public interface. A protocol can either consist of one message schema (template) or of the composition of two or more protocols. The notation is textual.
– **Integrated development environment**: Not available.
– **Parametricity**: BSPL relies upon parameters which can be adorned for capturing constraints on what parameter bindings to propagate in what direction. Splee (Chopra, Christie V, and Singh, 2017) generalizes BSPL by making roles themselves information parameters that take agents as values, paving the way to dynamic role binding and multicast.
– **Probability**: No paper has been published on probabilistic reasoning. Even though, Chopra and Singh consider that it can be supported and conceptually doable.
– **Testing/simulation**: an approach for engineering BSPL-based systems is

presented in (Singh, 2014) where Bliss, a conceptual model overlaying BSPL, is presented. Bliss yields simple steps to help ensure that the resulting protocol adequately captures the given requirements with respect to the social object. Even if testing is not explicitly mentioned in that paper, Bliss moves some initial steps towards this direction.
– **A priori verification**: in (Singh, 2012) the semantic requirements of BSPL are captured in a purely declaratively fashion, allowing a logic-based reasoner to compute with them.
– **Runtime verification**: BSPL and LoST, the Local State Transfer described in (Singh, 2011b) for enacting communication protocols following a declarative approach, go hand in hand. LoST is perfectly distributed and relies only upon the local knowledge of each business partner. It provides runtime verification in that each agent can verify the integrity of incoming messages. This verification is necessarily limited to what is detectable from the local view of the recipient.
– **Self-adaptation**: Not supported.
– **Protocol enactability and protocol-driven code generation**: BSPL is designed to enforce protocols to be enactable. Architectural adapters (local algorithms) for producing compliant BSPL enactments are presented in (Singh, 2011b) and have been implemented by Chopra's students.
– **Case studies and applications**: the NetBill protocol implementation in both BSPL and Bliss is presented in (Singh, 2014); more examples can be found in the other referenced works.

5.3 Commitment-based Interaction

This section has been validated by Amit Chopra and Munindar P. Singh.

Commitment Machines (CM), first introduced by Yolum and Singh in 2001 to describe agent interaction protocols in terms of the social commitments of the participants to one another (Yolum and Singh, 2001), spun off an impressive body of work and are one of the most lively paradigms for protocol modeling.

– **Modelled issues**: interaction protocols from a global perspective.
– **Modeling approach**: in the original paper, the formal language for representing commitment machines is based on propositional logic plus a "commitment" operator and a "leads to" operator to capture strict implication. Many languages based on social commitments and inspired by that seminal work exist; we may cite 2CL featuring the definition of patterns of interaction as sets of constraints (Baldoni, Baroglio, and Marengo, 2010); extensions that improve the way commitments are discharged and pre-conditions are specified (Winikoff, Liu, and Harland, 2004a), axiomatizations of operations in a first order Event Calculus framework to increase expressiveness (Chesani et al., 2013).
– **Integrated development environment**: a set of integrated software tools

supporting 2CL protocol design and analysis is presented in (Baldoni et al., 2014).
– **Parametricity**: in (Fornara and Colombetti, 2003), the content and the condition fields of commitment objects are described through schemes representing temporal proposition objects in parametric form; parametricity is also addressed in (Desai, Chopra, and Singh, 2009).
– **Probability**: In (Günay, Liu, and Zhang, 2016) Gunay, Liu and Zang present the ProMoca framework, which provides an expressive modeling language that includes various features to model commitment protocols. ProMoca supports probabilistic modeling to capture uncertainty in behaviours and beliefs of agents.
– **Testing/simulation**: tools supporting some steps (including testing) of the Amoeba methodology (Desai, Chopra, and Singh, 2009) for modeling and evolving commitment protocols exist (Desai et al., 2005a).
– **A priori verification**: many proposals for verifying generic properties and for model checking commitment protocols exist including (Desai et al., 2007b; El Kholy et al., 2014; El-Menshawy, Bentahar, and Dssouli, 2011; Yolum, 2006).
– **Runtime verification**: One of the first papers dealing with runtime verification of commitment-based protocols is (Venkatraman and Singh, 1999) where a vector representation of time and an early form of causality (e.g., making sure the delegation of a commitment is propagated to the creditor) are considered. More recently, the extension to commitment machines discussed in (Chesani et al., 2013) comes along with an implementation to support runtime monitoring, while Cupid (Chopra and Singh, 2015a) and Custard (Chopra and Singh, 2016) give information-based characterization of norms, including commitments; both support tracking runtime compliance with norms as the formalization is based on database queries.
– **Self-adaptation**: in (Dalpiaz et al., 2010), the authors formalize the notion of a participant's strategy for a goal in terms of the required commitments and present a conceptual model for adaptation built around this notion of strategy that allows using arbitrary strategy selection criteria.
– **Protocol enactability and protocol-driven code generation**: the necessary and sufficient conditions for enactability of commitment protocols are discussed in (Desai and Singh, 2008), where an algorithm for generating roles consistent with enactable protocols is also presented. Tosca (King et al., 2017) gives an operationalization of commitments over BSPL from the point of view of decentralized enactments.
– **Case studies and applications**: almost all the papers in the commitment protocols research strand present examples of use and case studies. Commitments have been used to real foreign exchange protocols and a real insurance protocol in Amoeba (Desai et al., 2007a), and there is one application outside academia carried out within the US Ocean Observatories Initiative and presented in (Arrott et al., 2009).

5.4 New Hierarchical Agent Protocol Notation

This section has been validated by Michael Winikoff, Nitin Yadav, and Lin Padgham.

HAPN, the New Hierarchical Agent Protocol Notation proposed in (Winikoff, Yadav, and Padgham, 2018), extends Finite State Machines into hierarchical FSM to cope with states that contain one or more concurrent sub-protocols.

– **Modelled issues**: interaction protocols from a global perspective.
– **Modeling approach**: protocols are modelled using a graphical notation that extends the standard FSM one by structuring each transition to follow the form `Sender → Receiver: msg(args) [guard]/effect`, showing for each (sub-)protocol its name and interface variables in the initial state, and allowing states to have sub-protocols. The bigger difference w.r.t. FSM, however, is not in the format of edges, but the fact that HAPN uses a hierarchical FSM with a particular semantics discussed in the paper.
– **Integrated development environment**: a prototype tool to create and edit protocols is presented in (Yadav, Padgham, and Winikoff, 2015a).
– **Parametricity**: the notation allows protocols to be parametrized by roles and variables.
– **Probability**: Not supported.
– **Testing/simulation**: the editing tool presented in (Yadav, Padgham, and Winikoff, 2015a) is able to simulate protocol execution.
– **A priori verification**: the importance verifying the protocol properties, including enactability, is clear to the HAPN authors who suggest that *"since HAPN protocols can be flattened into FSM with variables, and verification techniques use structures similar to FSMs, the existing techniques are applicable to verifying these properties in the context of HAPN protocols"*. The flattening algorithm is not implemented.
– **Runtime verification**: a runtime verification tool is not available, but the IDE supporting HAPN could easily be extended to check traces at runtime, since it is already able to check traces at design-time.
– **Self-adaptation**: Not supported.
– **Protocol enactability and protocol-driven code generation**: HAPN does not solve the enactability issue.
– **Case studies and applications**: in (Winikoff, Yadav, and Padgham, 2018) three case studies are presented: the play date (based on a real application), an auction, and holonic manufacturing.

5.5 Self-adaptive Systems Engineering

This section has been validated by Danny Weyns.

In a bunch of papers spanning from 2004 to now, Weyns and colleagues address the issues of engineering self-adaptive situated MAS from almost any

point of view, ranging from design (Iglesia and Weyns, 2015; Steegmans et al., 2004; Weyns, Schelfthout, and Holvoet, 2005) to static verification (Iftikhar and Weyns, 2012), from runtime verification (Iftikhar and Weyns, 2016) to simulation (Weyns and Iftikhar, 2016). As stated in (Weyns, 2012), the purpose of their research is to create a methodological approach and framework for (1) model checking of the behavior of a self-adaptive system during design, (2) model-based testing of the concrete implementation during development, and (3) runtime diagnosis after system deployment.

Although these works constitute a coherent body of knowledge, they cannot be sorted out into a unique approach like the others discussed in this section. Nevertheless, the findings presented therein are important for the MAS community, making the analysis of this approach extremely relevant.

– **Modelled issues**: all the design artifacts which characterize a self-adaptive system including architecture, control flow, actions, interactions.
– **Modeling approach**: depends on the work; as an example, in (Iftikhar and Weyns, 2012) the authors use timed automata to model the main processes in the system and timed computation tree logic for the specification of the required properties, whereas in (Weyns and Iftikhar, 2016) they use stochastic timed automata to describe runtime models of the system and runtime simulation for the analysis of quality properties.
– **Integrated development environment**: the ActivFORMS runtime environment (Active FORmal Models for Self-adaptation) (Iftikhar and Weyns, 2017), supported by the Uppaal suite (David et al., 2015), allows designers to model and verify a feedback loop. The verified models can be directly deployed on top of a runtime environment that executes them, supporting visualization of the executing models, the state of the goals, and on the fly updates of both.
– **Parametricity**: parametricity is supported in different ways depending on the modelled issue and the adopted formalism; for example, when stochastic timed automata models are used like in (Weyns and Iftikhar, 2016), they can be parametrized to capture variations, changes, and in particular uncertainties in the system or the environment.
– **Probability**: Weyns and colleagues (Calinescu et al., 2018; Weyns et al., 2018) use probabilistic models at runtime to represent a system and its environment from the point of view of different quality properties (for example a model that represents energy consumption of an IoT network, or a model that represents the communication latency in the network). These models have two types of parameters. On the one hand, the models have parameters that represent uncertainties of the system or its environment (for example the interference of network links). These parameters are updated with actual values (possibly learned over time). On the other hand, the models have parameters that represent “knobs” that can be set to model different possible system configurations (for example the power setting of the radio of a node to communicate messages over a wireless link to another node). They then apply (statistical) model checking at runtime to analyse these models and predict the expected quality

properties of different configurations. Based on the results they may then adapt the system to a new configuration (i.e., the one with the best predicted values for a set of quality goals) to ensure or improve its quality goals.
– **Testing/simulation**: ActivFORMS allows users to perform a set of tests aimed at validating their models before deployment or selecting adaptation options during runtime.
– **A priori verification**: exploitation of model checking to verify behavioral properties of decentralized self-adaptive systems has been faced in many works including (Iftikhar and Weyns, 2012).
– **Runtime verification**: in one of the most recent works dealing with verification of properties at runtime (Iftikhar and Weyns, 2016), the authors exploit runtime statistical model checking as an efficient strategy to tradeoff between the accuracy of the provided guarantees and the required computation time and system resources.
– **Self-adaptation**: all the works mentioned in this section take self-adaptation into account.
– **Protocol enactability and protocol-driven code generation**: Not supported.
– **Case studies and applications**: many self-adaptive systems artifacts and model problems exemplars have been developed by the authors including (Gerasimou et al., 2017; Iftikhar et al., 2017; Weyns and Calinescu, 2015). The artifact presented in (Iftikhar et al., 2017) offers a real world deployment of an IoT setup that can be used for experimentation. The IoT network has been developed for the VersaSense company, `https://www.versasense.com/`.

Part III

Formalism extensions

This part of the thesis is totally focused on the formalism extensions that have been proposed during the Ph.D. program.
The first extension is presented in Chapter 6. It brings variables inside the trace expression formalism. Thanks to such extension, we can represent more complex properties and protocols, because we can handle events containing values. In Chapter 6, we show how the trace expressions semantics can be extended with variables and we motivate our work through examples.
The second extension is presented in Chapter 7. It brings probabilities inside the trace expression formalism. Even though the motivations related to this extension are less intuitive than the motivations for the parametric extensions, in Chapter 7 we show how introducing probabilities inside our formalism can increment its robustness for real world scenarios, where we are interested in verifying less reliable systems (with high uncertainty in the observed events).
After the presentation of the probabilistic extension, we will present how to use it in order to achieve the runtime verification of systems with state estimation.

6 *Parametric Trace Expressions*

"Try not to become a man of success.
Rather become a man of value."
- Albert Einstein

In this chapter, we propose an extension of the trace expression formalism to support parametric runtime verification. The full semantics of parametric trace expressions is presented, together with some examples to show their expressive power. The proposed extension has been implemented and experimented in JADE with a non trivial case study consisting of a variation of the English auction interaction protocol specification.

The contents of this chapter are published in (Ancona, Ferrando, and Mascardi, 2017)

6.1 Introduction

During the Ph.D. program, there have been several occasions where we needed a more expressive formalism to properly manipulate properties and protocols. Since it is common the use of parameters inside specifications to handle datas, times, and so on, our first attempt to extend the trace expression formalism has been the addition of parameters in event types. As we will see after in the chapter, even though such formalism extension is not too much invasive, it allows enlarging undoubtedly the set of possible properties and protocols that can be defined.

In particular, *parametricity* (Luo et al., 2014b) is an important feature of a monitoring system for making RV more effective, since, typically, correctness of traces depends on the specific data values that are carried by the monitored events of the trace, and that, in general, cannot be predicted statically. For instance, the verification of an interaction protocol may require that the values exchanged by two agents are in a certain relation; protocols may also be parametric in the involved agents, and resources, and this parametricity is naturally reflected on the data carried by values.

6.1.1 Illustrative example

In Section 4.2.2, we presented the general notion of event type. In order to make the presentation of the parametric extension more clear, we define a predicate *match* to represent when an event matches a specific event type.

$$match(e, \vartheta) \iff e \in [\![\vartheta]\!]$$

Considering again the ping pong example introduced in Section 4.3.1, where there were the event types *ping* and *pong*, we now have:

$$match(e, ping) \iff e \in [\![ping]\!] \iff e = send(alice, bob, _)$$

$$match(e, pong) \iff e \in [\![pong]\!] \iff e = send(bob, alice, _)$$

Even though the ping-pong protocol is extremely simple, it allows us to introduce intuitively the advantage of parametricity. In particular, considering the infinite version *PingPongForever*, we could be interested in enriching the events' behaviour with values. We may add an integer to the *ping* and *pong* communications, with the constraint that *alice* sends an integer to *bob*, and *bob* must reply with a greater integer.

For instance, a valid execution trace could start with $send(alice, bob, inform(42))$ followed by $send(bob, alice, inform(43))$, $send(alice, bob, inform(2))$, and $send(bob, alice, inform(5))$.

Let $ping(_)$ and $pong(_)$ denote the event types s.t.

$$match(e, ping(n)) \text{ iff } n \in \mathbb{Z}, e = send(alice, bob, inform(n))$$

and

$$match(e, pong(n)) \text{ iff } n \in \mathbb{Z}, e = send(bob, alice, inform(k)) \text{ and } k > n$$

With these event types, the best specification we can write with a trace expression is

$$PingPongForever_2 = ping(n){:}pong(n){:}PingPongForever_2$$

However, this specification fails to support data parametricity since the value n has to be picked up once for all, hence the trace expression defines a protocol where at each round *alice* has to send the same integer.

The problem becomes even more evident if we consider the ping-pong protocol which requires both agents to always reply with an integer greater than the previously received one. For instance, the trace shown above would not be correct, while a valid execution trace could start with

$$send(alice, bob, tell(42))$$

followed by

$$send(bob, alice, inform(43)), send(alice, bob, inform(52)),$$

and

$$send(bob, alice, tell(55)).$$

6.2 Formalization

To support parametricity, we let event types in trace expressions to contain variables; following the Prolog notation, we use identifiers starting with a capital letter to denote variables in event types; for instance, event type $ping(N)$ contains variable N. Accordingly, the semantics of event types and, hence, the *match* function, have to be extended; if ϑ is an event type, possibly containing free variables, then we write $match(e, \vartheta) = \sigma$ to mean that event e matches event type ϑ with computed substitution σ which must be grounding for the event type ϑ, that is, the domain on which σ is defined coincides with the set of variables in ϑ.

Substitutions are finite domain partial maps from a fixed universe of variables $\mathcal{X}$ into a fixed universe of values $\mathcal{V}$; we denote with $dom(\sigma)$ the finite domain of σ. The substitution with the empty domain is denoted by $\emptyset$. The equality $\sigma = \sigma_1 \cup \sigma_2$ holds iff $dom(\sigma) = dom(\sigma_1) \cup dom(\sigma_2)$, and for all $X \in dom(\sigma)$, $\sigma(X) = \sigma_1(X)$ if $X \in dom(\sigma_1)$, and $\sigma(X) = \sigma_2(X)$ if $X \in dom(\sigma_2)$ (hence, σ_1 and σ_2 must coincide on $dom(\sigma_1) \cap dom(\sigma_2)$). The notation $\sigma_{|X}$ denotes the substitution where X is removed from its domain: $\sigma_{|X} = \sigma'$ iff $dom(\sigma') = dom(\sigma) \setminus \{X\}$ and for all $X \in dom(\sigma')$ $\sigma'(X) = \sigma(X)$. The notation $\sigma\vartheta$ denotes the event type obtained from ϑ by substituting all occurrences of $X \in dom(\sigma)$ in ϑ with $\sigma(X)$.

Besides extending event types with variables, we need to introduce a new trace expression construct to control the scope of variables: $<X; \tau>$ binds the free occurrences of X in τ; accordingly, the trace expression $\sigma\tau$ obtained from τ by substituting all free occurrences of $X \in dom(\sigma)$ in τ with $\sigma(X)$, is defined as follows:

$$\sigma(\vartheta{:}\tau) = (\sigma\vartheta){:}(\sigma\tau)$$
$$\sigma(\tau_1\, op\, \tau_2) = (\sigma\tau_1)\, op\, (\sigma\tau_2) \text{ for } op \in \{\vee, \wedge, |, \cdot\}$$
$$\sigma(<X; \tau>) = <X; \sigma_{|X}\tau>$$

The transition system for parametric trace expressions is defined in Figure 6.1.

$$\text{(main)}\frac{\tau \stackrel{e}{\rightarrowtail} \tau';\emptyset}{\tau \stackrel{e}{\rightarrow} \tau'} \qquad \text{(prefix)}\frac{}{\vartheta{:}\tau \stackrel{e}{\rightarrowtail} \tau;\sigma}\ \sigma{=}match(e,\vartheta) \qquad \text{(or-l)}\frac{\tau_1 \stackrel{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1{\vee}\tau_2 \stackrel{e}{\rightarrowtail} \tau_1';\sigma}$$

$$\text{(or-r)}\frac{\tau_2 \stackrel{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1{\vee}\tau_2 \stackrel{e}{\rightarrowtail} \tau_2';\sigma} \qquad \text{(and)}\frac{\tau_1 \stackrel{e}{\rightarrowtail} \tau_1';\sigma_1 \quad \tau_2 \stackrel{e}{\rightarrowtail} \tau_2';\sigma_2}{\tau_1{\wedge}\tau_2 \stackrel{e}{\rightarrowtail} \tau_1'{\wedge}\tau_2';\sigma}\ \sigma{=}\sigma_1{\cup}\sigma_2 \qquad \text{(shuffle-l)}\frac{\tau_1 \stackrel{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1|\tau_2 \stackrel{e}{\rightarrowtail} \tau_1'|\tau_2;\sigma}$$

$$\text{(shuffle-r)}\frac{\tau_2 \stackrel{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1|\tau_2 \stackrel{e}{\rightarrowtail} \tau_1|\tau_2';\sigma} \qquad \text{(cat-l)}\frac{\tau_1 \stackrel{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1{\cdot}\tau_2 \stackrel{e}{\rightarrowtail} \tau_1'{\cdot}\tau_2;\sigma} \qquad \text{(cat-r)}\frac{\tau_2 \stackrel{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1{\cdot}\tau_2 \stackrel{e}{\rightarrowtail} \tau_2';\sigma}\ \epsilon(\tau_1)$$

$$\text{(var-t)}\frac{\tau \stackrel{e}{\rightarrowtail} \tau';\sigma}{<X;\tau> \stackrel{e}{\rightarrowtail} \sigma\tau';\sigma_{|X}}\ X{\in}dom(\sigma) \qquad \text{(var-f)}\frac{\tau \stackrel{e}{\rightarrowtail} \tau';\sigma}{<X;\tau> \stackrel{e}{\rightarrowtail} <X;\tau'>;\sigma}\ X{\notin}dom(\sigma) \qquad (\epsilon\text{-var})\frac{\epsilon(\tau)}{\epsilon(<X;\tau>)}$$

FIGURE 6.1. Transition system for parametric trace expressions

The main transition relation $\tau \stackrel{e}{\rightarrow} \tau'$ is defined in terms of the auxiliary relation $\tau \stackrel{e}{\rightarrowtail} \tau';\sigma$, where σ is the substitution generated during the transition step. This is required because it is not possible to predict from which variable occurrence a certain substitution is generated; consider for instance the trace expressions $\vartheta_1(X){:}\tau_1|\vartheta_2(X){:}\tau_2$, where $\vartheta_1(X)$ and $\vartheta_2(X)$ are two distinct event types containing occurrences of variable X; if event e fires a transition on the lhs operand of the shuffle operator, then the computed substitution σ is s.t. $\sigma = match(e,\vartheta_1)$, and the trace expression is rewritten into $\sigma(\tau_1|\vartheta_2(X){:}\tau_2)$, otherwise, if the transition is fired on the rhs, then $\sigma = match(e,\vartheta_2)$, and the trace expression is rewritten into $\sigma(\vartheta_1(X){:}\tau_1|\tau_2)$.

Rule (main) defines the main transition relation in terms of the auxiliary transition relation which computes the substitution; at top level a correct trace expression cannot contain free variables (that is, undeclared variables), hence the main transition is fired only if the computed substitution is empty.

In rule (prefix) the substitution is generated by the *match* function applied to the current event e and the event type ϑ.

Rules for union, shuffle, and concatenation are the corresponding generalization of the rules in Figure 4.1.

In rule (and) the side condition requires that the substitutions σ_1 and σ_2 computed for the two operands must coincide on the intersection of their domains; the final substitution σ is obtained by merging σ_1 and σ_2. For instance, for the trace expression

$$\tau = send(alice,R,C){:}\tau_1{\wedge}send(S,bob,C){:}\tau_2$$

and the event e s.t. Alice sends to Bob *tell*(42), we have

$$send(alice,R,C){:}\tau_1 \stackrel{e}{\rightarrowtail} \tau_1;\{R\mapsto bob, C\mapsto tell(42)\}$$
$$send(S,bob,C){:}\tau_2 \stackrel{e}{\rightarrowtail} \tau_2;\{S\mapsto alice, C\mapsto tell(42)\}$$

therefore $\tau \stackrel{e}{\rightarrowtail} \tau_1{\wedge}\tau_2;\{R\mapsto bob, S\mapsto alice, C\mapsto tell(42)\}$.

Rules (var-t) and (var-f) deal with the new construct $<X;\tau>$ for variable scoping. The former rule is applied when variable X is contained in the domain

of the computed substitution σ for the transition starting from τ; in such a case σ is applied to the trace expression τ' in which τ rewrites to, and the scoping construct is removed; furthermore, the computed substitution is $\sigma_{|X}$. The latter rule is applied when variable X is not contained in the domain of the computed substitution σ for the transition starting from τ; in this case the scoping construct is not removed, and the computed substitution coincides with σ.

Rules for the auxiliary predicate $\epsilon(_)$ are the same as those in Figure 4.1, except for the straightforward rule for the new construct $<X;\tau>$.

6.3 Illustrative example revisited

Let us consider again the variation of the $PingPongForever_2$ protocol proposed in Section 6.1.1, where Alice sends an integer to Bob, and Bob must reply with a greater integer; such a protocol can be specified by the following parametric trace expressions:

$$PingPongForever_3 = \\ <N; ping(N){:}pong(N){:}PingPongForever_3>$$

where, again, $ping(_)$ and $pong(_)$ denote the event types s.t. $match(e, ping(n))$ iff $n \in \mathbb{Z}$, $e = send(alice, bob, tell(n))$, and $match(e, pong(n))$ iff $n \in \mathbb{Z}$, $e = send(bob, alice, tell(k))$ and $k > n$, respectively.

Since $PingPongForever_3$ does not contain free variables, applying the substitution $\{N \mapsto 42\}$ to the trace expression $pong(N){:}PingPongForever_3$ yields the trace expression $pong(42){:}PingPongForever_3$.

Let e_1 and e_2 correspond to the events "Alice sends $tell(42)$ to Bob", and "Bob sends $tell(45)$ to Alice", respectively; we show that the steps $\tau \xrightarrow{e_1} pong(42){:}\tau \xrightarrow{e_2} \tau$ are derivable, where τ is the trace expression defined by $PingPongForever_3$.

The transition step $\tau \xrightarrow{e_1} pong(42){:}\tau$ can be derived with the following derivation tree, since $match(e_1, ping(N)) = \{N \mapsto 42\}$:

$$\text{(main)}\dfrac{\text{(var-t)}\dfrac{\text{(prefix)}\dfrac{}{ping(N){:}pong(N){:}\tau \overset{e_1}{\rightarrowtail} pong(N){:}\tau; \{N \mapsto 42\}}}{<N; ping(N){:}pong(N){:}\tau> \overset{e_1}{\rightarrowtail} \{N \mapsto 42\}(pong(N){:}\tau); \emptyset}}{<N; ping(N){:}pong(N){:}\tau> \xrightarrow{e_1} pong(42){:}\tau}$$

The further transition step $pong(42){:}\tau \xrightarrow{e_2} \tau$ can be derived as follows, since $match(e_2, pong(42)) = \emptyset$:

$$\text{(main)}\dfrac{\text{(prefix)}\dfrac{}{pong(42){:}\tau \overset{e_2}{\rightarrowtail} \tau; \emptyset}}{pong(42){:}\tau \xrightarrow{e_2} \tau}$$

With this second transition step the first round of the protocol is completed and the current state of the system is represented again by τ (that is, $PingPongForever_3$); hence, Alice is expected to send any integer value to Bob (and not just 42), as specified by the protocol.

We can now consider the more challenging version of the protocol where both Alice and Bob are required to reply with an integer greater than the

previously received one. In this case we need to adopt a pattern that will be employed also in the case study in Section 6.4 to propagate information regarding data values; in this particular case, the last received integer is propagated with two different variables N_1 and N_2 whose use is alternated in the transition steps: at one transition step N_1 and N_2 carry the last sent value and the previous one, respectively, while at the next transition step N_1 and N_2 carry the previously sent value and the last one, respectively.

$$
\begin{aligned}
&\mathit{PingPongForever}_4 = {<}N_1; \mathit{ping}(N_1){:}\mathit{Forever}{>}\\
&\mathit{Forever} = {<}N_2; \mathit{pong}(N_1, N_2){:}{<}N_1; \mathit{ping}(N_2, N_1){:}\mathit{Forever}{>}{>}
\end{aligned}
$$

Besides the event type $\mathit{ping}(_)$ already used in the previous examples, we employ also the event types $\mathit{ping}(_, _)$, and $\mathit{pong}(_, _)$ s.t. $\mathit{match}(e, \mathit{ping}(k, n))$ iff $k, n \in \mathbb{Z}$, $e = \mathit{send}(\mathit{alice}, \mathit{bob}, \mathit{tell}(n))$, $k < n$, and $\mathit{match}(e, \mathit{pong}(n))$ iff $k, n \in \mathbb{Z}$, $e = \mathit{send}(\mathit{bob}, \mathit{alice}, \mathit{tell}(n))$ and $k < n$, respectively.

Let e_1, e_2, and e_3 correspond to the events “Alice sends $\mathit{tell}(42)$ to Bob”, “Bob sends $\mathit{tell}(45)$ to Alice”, and “Alice sends $\mathit{tell}(46)$ to Bob”, respectively; we show that the steps $\tau \xrightarrow{e_1} \tau_1 \xrightarrow{e_2} \tau_2 \xrightarrow{e_3} \tau_3$ are derivable, for suitable τ_1, τ_2, and τ_3, with τ denoting the trace expression defined by $PingPongForever_4$.

If $\tau_1 = {<}N_2; \mathit{pong}(42, N_2){:}{<}N_1; \mathit{ping}(N_2, N_1){:}\mathit{Forever}{>}{>}$, then the transition step $\tau \xrightarrow{e_1} \tau_1$ can be derived with the following derivation tree, since $\mathit{match}(e_1, \mathit{ping}(N_1)) = \{N_1 \mapsto 42\}$:

$$
\text{(main)}\dfrac{\text{(var-t)}\dfrac{\text{(prefix)}\dfrac{}{\mathit{ping}(N_1){:}\mathit{Forever} \overset{e_1}{\rightarrowtail} \mathit{Forever}; \{N_1 \mapsto 42\}}}{{<}N_1; \mathit{ping}(N_1){:}\mathit{Forever}{>} \overset{e_1}{\rightarrowtail} \{N_1 \mapsto 42\}\mathit{Forever}; \emptyset}}{{<}N_1; \mathit{ping}(N_1){:}\mathit{Forever}{>} \xrightarrow{e_1} \tau_1}
$$

The next transition step $\tau_1 \xrightarrow{e_2} \tau_2$ can be derived with $\tau_2 = {<}N_1; \mathit{ping}(45, N_1){:}\mathit{Forever}{>}$, since $\mathit{match}(e_2, \mathit{pong}(42, N_2)) = \{N_2 \mapsto 45\}$.

$$
\text{(main)}\dfrac{\text{(var-t)}\dfrac{\text{(prefix)}\dfrac{}{\tau_1' \overset{e_2}{\rightarrowtail} {<}N_1; \mathit{ping}(N_2, N_1){:}\mathit{Forever}{>}; \{N_2 \mapsto 45\}}}{\tau_1 \overset{e_2}{\rightarrowtail} {<}N_1; \mathit{ping}(45, N_1){:}\mathit{Forever}{>}; \emptyset}}{\tau_1 \xrightarrow{e_2} \tau_2}
$$

In the derivation tree,

$$\tau_1' = \mathit{pong}(42, N_2){:}{<}N_1; \mathit{ping}(N_2, N_1){:}\mathit{Forever}{>}.$$

Finally, the transition step $\tau_2 \xrightarrow{e_3} \tau_3$ can be derived with

$$\tau_3 = {<}N_2; \mathit{pong}(46, N_2){:}{<}N_1; \mathit{ping}(N_2, N_1){:}\mathit{Forever}{>}{>}$$

since $\mathit{match}(e_3, \mathit{ping}(45, N_1)) = \{N_1 \mapsto 46\}$.

$$
\text{(main)}\dfrac{\text{(var-t)}\dfrac{\text{(prefix)}\dfrac{}{\mathit{ping}(45, N_1){:}\mathit{Forever} \overset{e_3}{\rightarrowtail} \mathit{Forever}; \{N_1 \mapsto 46\}}}{\tau_2 \overset{e_3}{\rightarrowtail} \{N_1 \mapsto 46\}\mathit{Forever}; \emptyset}}{\tau_2 \xrightarrow{e_3} \tau_3}
$$

6.4 Case study

In this section we show how parametric trace expressions support runtime monitoring of a non trivial interaction protocol between agents.

In particular, we formalize through parametric trace expressions an English auction where the auctioneer proposes to sell an item for a given price and the bidders either accept or reject the proposal; as long as more than one bidder accepts, the price is raised and another negotiation round is made. The protocol is consistent with the existing descriptions of the English auction that can be found online, even if it slightly differs from the English auction FIPA specification (Foundation for Intelligent Physical Agents, 2001).

6.4.1 Informal specification of the protocol

The protocol involves an *auctioner* agent, and two or more *bidder* agents. The protocol starts with a preamble where *auctioner* sends a single message to all bidders (in any order) informing that the auction for selling a certain item is going to start. It is assumed that all contacted bidders will participate to the auction, until *auctioner* will notify bidders about the closing of the auction.

The preamble is followed by an initial proposal round, where *auctioner* sends a single message to all bidders with a proposed item price, which is equal for all bidders; then *auctioner* waits to receive a single reply from all bidders before deciding to either closing the auction, or moving to the next proposal round. Every bidder can either accept or reject the proposal; a bidder b will keep attending the auction, even when b decides to reject a proposal at a certain round.

Agent *auctioner* moves to the next proposal round if at least two bidders have accepted the previously proposed price; in such a case, the new proposed price will be at least greater or equal than $p + \Delta p$, where p is the price proposed at the previous round, and Δp is an a priori fixed positive constant.

If one bidder (let us assume it is bidder b) has accepted the last proposed price, then *auctioner* closes the auction by sending a single message to all bidders; bidder b is notified of the purchase of the item, while all other bidders are informed that the item has been sold.

Finally, if no bidder accepts the current proposal, then *auctioner* closes the auction by sending a single message to all bidders to notify them that the item is unsold.

6.4.2 Formal specification of the protocol

For simplifying the presentation we describe the parametric trace expression specifying the protocol for just three agents: the *auctioner* and two bidders $bidder_1$ and $bidder_2$; it is possible to generalize such a specification[1] to monitor

[1] The generalized trace expression is described at `http://www.ParametricTraceExpr.altervista.org`, from where the Prolog specification of all the examples presented in

a system with an unspecified number of participants that can be fixed only at runtime.

Even in the simplified setting, parametric trace expressions are still required for checking the following facts: at each proposal round the same proposed price is sent to all bidders; at each proposal round *auctioner* proposes a price which has been increased of at least Δp, if there has been a previous round; the auction is correctly closed with the expected notifications to bidders.

We provide the semantics of the protocol ground event types only, as the semantics for non-ground event types can be easily derived from it. For instance, $match(e, start(b))$ iff $e = send(auctioner, b, tell(start))$ just means that, for the ground event type $start(b)$, matching with event e succeeds and returns the empty substitution if and only if $e = send(auctioner, b, tell(start))$ (that is, *auctioner* sends to b the message with content $tell(start)$).

Consequently, $match(e, start(B))$ succeeds and returns the substitution $\{B \mapsto b\}$ iff $match(e, start(b))$ succeeds (with the empty substitution).

In the following, we use the _ anonymous variable typical of Prolog to represent part of the term we are not interested in. With anonymous variables we unify anything (as for free variables but without the need of a name since we will not use it).

$match(e, start)$ iff $e = send(auctioner, _, tell(start))$
$match(e, start(b))$ iff $e = send(auctioner, b, tell(start))$
$match(e, is_bidder(b))$ iff
  $e = send(auctioner, b, _)$ or $e = send(b, auctioner, _)$
$match(e, prop)$ iff $e = send(auctioner, _, propose(_))$
$match(e, prop\$(p))$ iff $e = send(auctioner, _, propose(price(p)))$
$match(e, prop_to(b))$ iff $e = send(auctioner, b, propose(_))$
$match(e, prop(b, p))$ iff $e = send(auctioner, b, propose(price(p)))$
$match(e, prop(b, p_1, p_2))$ iff
  $e = send(auctioner, b, propose(price(p_2))), p_2 \geq p_1 + \Delta p$
$match(e, yes)$ iff $e = send(_, auctioner, accept-proposal)$
$match(e, yes(b))$ iff $e = send(b, auctioner, accept-proposal)$
$match(e, no)$ iff $e = send(_, auctioner, reject-proposal)$
$match(e, no(b))$ iff $e = send(b, auctioner, reject-proposal)$
$match(e, reply)$ iff $match(e, yes)$ or $match(e, no)$
$match(e, fail)$ iff $e = send(auctioner, _, tell(close(fail)))$
$match(e, fail(b))$ iff $e = send(auctioner, b, tell(close(fail)))$
$match(e, buy(b))$ iff $e = send(auctioner, b, tell(close(buy)))$
$match(e, close)$ iff $e = send(auctioner, _, tell(close(_)))$
$match(e, prop_or_reply)$ iff $match(e, prop)$ or $match(e, reply)$
$match(e, reply_or_close)$ iff $match(e, reply)$ or $match(e, close)$

We use the filter and intersection operators to decompose the specification modularly in order to simplify definitions and improve readability. The para-

this chapter, the parametric trace expressions transition function, and the JADE MAS and monitor can be downloaded.

metric trace expression specifying the overall protocol can be obtained as the intersection of the following simpler parametric trace expressions, each enforcing some of the properties that have to be verified by the protocol. Adopting this divide et impera specification pattern helps the protocol designer to cope with complex protocols, by modeling one property at a time.

In the rest of the chapter we will abbreviate with ϑ trace expressions of shape $\vartheta{:}\epsilon$, when such a shortcut will not arise ambiguities.

PREAMBLE. The protocol must start with two messages of *auctioner* sent to all bidders (in any order) to inform them that the auction is going to start.

$$Preamble = (start(bidder_1)|start(bidder_2)){\cdot}1$$

The use of the derived constant 1 (see Section 4.2.6) denoting the set of all possible traces corresponds to the fact that this trace expression specifies the preamble only.

BIDDER FLOW. Trace expressions $Bidder_1$ and $Bidder_2$ specify the correct message flow for $bidder_1$ and $bidder_2$, respectively.

$$\begin{aligned}
Bidder_1 &= is_bidder(bidder_1) \gg Bidder \\
Bidder_2 &= is_bidder(bidder_2) \gg Bidder \\
Bidder &= start{:}prop{:}reply{:}BidderLoop \\
BidderLoop &= close \vee prop{:}reply{:}BidderLoop
\end{aligned}$$

Through the $\gg$ operator, only messages involving either $bidder_1$ (for trace expression $Bidder_1$) or $bidder_2$ (for trace expression $Bidder_2$) are checked. Thanks to the use of the filter operator, the two trace expressions $Bidder_1$ and $Bidder_2$ can share the same trace expression *Bidder* (whose definition depends from *BidderLoop*).

Among all requirements imposed by these trace expressions there is also the constraint that there must exist at least one proposal round.

SAME PRICE PROPOSED TO BIDDERS. This parametric trace expression requires that at each round *auctioner* must send the same proposed price to bidders. Through the $\gg$ operator, *SamePrice* only checks messages which are proposals sent by *auctioner* to bidders.

$$\begin{aligned}
SamePrice &= prop \gg PriceLoop \\
PriceLoop &= \epsilon \vee {<}P;\, prop\$(P){:}prop\$(P){:}PriceLoop{>}
\end{aligned}$$

Since a new proposal round can be started only after all bidders have sent their replies (see the next trace expression), the two contiguous proposals that share the same price must have necessarily been sent to different bidders. The trace expression ϵ specifies that the sequence of proposal rounds is allowed to terminate.

NEXT PROPOSAL ROUND. The following trace expression specifies that *auctioner* can start a new proposal round only after having received replies from all bidders. Through the $\gg$ operator, *NextRound* only checks messages which are either proposals sent by *auctioner* to bidders, or acceptance/rejection

replies sent by bidders to *auctioner*.

$$
\begin{array}{l}
NextRound = prop_or_reply \gg RoundLoop \\
RoundLoop = \epsilon \lor \\
\quad prop_to(bidder_1){:}((prop_to(bidder_2)|reply|reply) \cdot RoundLoop) \lor \\
\quad prop_to(bidder_2){:}((prop_to(bidder_1)|reply|reply) \cdot RoundLoop)
\end{array}
$$

At each round, if the first proposal is sent to $bidder_1$, then the next proposal must be sent to $bidder_2$ (and the previous parametric trace expression ensures that the two proposals share the same price), and two replies have to be sent. The trace expressions $Bidder_1$ and $Bidder_2$ defined above guarantee that each reply must follow the corresponding proposal, hence the shuffle operator can be safely used in $prop_to(bidder_2)|reply|reply$. A symmetric trace expression deals with the case when the first proposal is sent to $bidder_2$. As for *SamePrice*, the trace expression ϵ specifies that the sequence of proposal rounds is allowed to terminate.

INCREASED PRICE. This parametric trace expression specifies that at each proposal round *auctioner* proposes a price which has been increased of at least Δp (an a priori fixed positive constant), if there has been a previous round. The constant Δp is implicitly defined in the event type $prop(b, p_1, p_2)$ which succeeds if price p_2 is proposed to bidder b, and $p_2 \geq p_1 + \Delta p$ (where p_1 is the price proposed at the previous round; see the event types semantics defined before).

This parametric trace expression is based on the same pattern used in $PingPongForever_4$ in Section 6.1.1.

Interestingly, by virtue of the parametric trace expression *SamePrice* that imposes that the same price is sent to bidders at each proposal round, it suffices to check that the price is correctly increased only for the proposals sent to one of the bidders; in this case the trace expression arbitrarily checks only the proposals sent to $bidder_1$ (through the $\gg$ operator), but of course checking the proposals sent to $bidder_2$ would work as well.

$$
\begin{array}{l}
IncPrice = prop_to(bidder_1) \gg <P_1; prop(bidder_1, P_1){:}IncLoop> \\
IncLoop = \epsilon \lor \\
\ <P_2; prop(bidder_1, P_1, P_2){:}<P_1; prop(bidder_1, P_2, P_1){:}IncLoop>>
\end{array}
$$

As for *SamePrice*, and *NextRound*, the trace expression ϵ specifies that the sequence of proposal rounds is allowed to terminate.

CLOSING. This parametric trace expression guarantees that the auction is correctly closed with the expected notifications to bidders. Only messages which are bidders reply or *auctioner* closing messages are checked through the $\gg$ operator.

While the property enforced with *SamePrice* and *IncPrice* can only be specified with parametric trace expressions, closing could be expressed with a non parametric trace expression; however, the use of variables makes the specification more compact and readable.

$$
\begin{array}{l}
Close = reply_or_close \gg CloseLoop \\
CloseLoop = \\
<B; yes(B){:}(yes{:}CloseLoop \lor <B'; no(B'){:}(buy(B)|fail(B'))>) \lor \\
\quad no(B){:}(no{:}fail{:}fail \lor <B'; yes(B'){:}(buy(B')|fail(B))>)>
\end{array}
$$

Variables B and B' are required for ensuring the correctness of closing messages sent by *auctioner*. The intersection of trace expressions $Bidder_1$, $Bidder_2$, and *NextRound* guarantees that two replies in the same round always originate from different bidders.

If bidder B accepts the proposal ($yes(B)$), then there are two possible cases: either the other bidder accepts the proposal as well (*yes*) and, hence, there will be another proposal round (*CloseLoop*), or the other bidder B' rejects the proposal ($no(B')$) and, hence, *auctioner* sends (in any order) the messages to the corresponding bidders notifying that B has purchased the item ($buy(B)$), and B' has not ($fail(B')$).

If bidder B rejects the proposal ($no(B)$), then there are two possible cases: either the other bidder rejects the proposal as well (*no*) and, hence, *auctioner* notifies both bidders that they have not purchased the item (again, trace expressions $Bidder_1$ and $Bidder_2$ guarantee that the two events matching *fail*:*fail* correspond to messages sent to different bidders); or the other bidder B' accepts the proposal ($yes(B')$) and, hence, *auctioner* sends (in any order) the messages to the corresponding bidders notifying that B' has purchased the item ($buy(B')$), and B has not ($fail(B)$).

PUTTING ALL TRACE EXPRESSIONS TOGETHER. Finally, the overall specification of the protocol can be obtained by assembling together the parametric trace expressions defined above through the intersection operator.

$$EnglishAuction = Preamble \wedge Bidder_1 \wedge Bidder_2 \wedge SamePrice \wedge NextRound \wedge IncPrice \wedge Close$$

6.5 Discussion

We have proposed parametric trace expressions, an extension to trace expressions expressly designed for parametric RV of multiagent systems.

Besides providing its formalization, we have implemented and experimented RV with parametric trace expressions in JADE with a non trivial case study consisting of a variation of the English auction interaction protocol specification.

Parametric trace expressions have been also integrated inside the IDE RIVERtools (Chapter 14). In this way, it is possible to define parametric protocols in an easy and compact way.

7 Probabilistic Trace Expressions

"If we find ourselves with a desire that nothing in this world can satisfy, the most probable explanation is that we were made for another world."
- C.S. Lewis

Probabilistic Trace Expressions (PTEs) are an extension of Trace Expressions where the types of events that can be observed by a monitor are associated with an observation probability. In this chapter we introduce PTEs, we adapt the runtime verification with state estimation approach proposed by Scott D. Stoller et al. in 2011 to them, and we present a semantics for PTEs that allows for the estimation of the probability to reach a given state, given a sequence of observations which may include observation gaps. Thanks to built-in operators suitable for checking that two different PTEs can perform the same transitions, and to algorithms for transforming LTL properties into PTEs, verifying that a LTL property is met by a system modeled by a PTE can be addressed in an elegant and natural way, even when gaps are observed.

The contents of this chapter will be submitted to a Journal in the MAS research area.

7.1 Introduction

Runtime verification of complex, distributed systems under ideal conditions (perfect observability of all the relevant events, no leaky communication channels, etc) is an hard task to perform, and has been addressed by many scientific works including surveys and introductory papers (Bartocci et al., 2018; Havelund, Reger, and Rosu, 2018; Leucker and Schallhart, 2009), books (Bartocci and Falcone, 2018), seminars (Bonakdarpour et al., 2016a; Havelund et al., 2010), and conferences[1]. When the conditions are not ideal and some relevant events cannot be observed by the monitor, generating a *gap* in the event trace, the problem becomes even harder (Babaee, Gurfinkel, and Fischmeister, 2018; Bartocci et al., 2011; Joshi, Tchamgoue, and Fischmeister, 2017). A gap represents the absence of information in the analyzed trace and corresponds to an execution point – or to a time slot – where the monitor does not know what the system did. Gaps may be due to the process of sampling observed events to reduce monitoring overhead, but also to events that are partially observable or not observable at all by the monitor: the monitor might be aware that an event took place, but does not know which. The introduction of gaps raises problems in checking that a temporal property is verified by the system, given that a trace of events (which may include gaps) has been observed. If the monitor does not know which event has been observed, it cannot know whether the temporal property is satisfied or not.

In (Stoller et al., 2011), each time a gap in observed a Hidden Markov Model (HMM) of the system is queried to know which events could be observed in the current state of the system, and with which probability. This allows the authors to estimate the probability to reach some state s_i after observing $obs = O_1, O_2, ..., O_t$ events, and – by generating a monitor that combines the system HMM and the temporal property ϕ into a single integrated model – to estimate the probability that ϕ is satisfied after observing $obs = O_1, O_2, ..., O_t$ events.

In this chapter we take (Stoller et al., 2011) as starting point, and we combine the approach presented therein with trace expressions, in order to obtain Probabilistic Trace Expressions.

Probabilistic Trace Expressions (PTEs) are an extension of Trace Expressions with probabilities associated with event types (Section 7.3). PTEs are more expressive than HMM, deterministic finite state machines and linear time temporal logic, being able to model more than context free languages. Besides sketching the *Probabilize* algorithm for transforming trace expressions into PTEs in Section 7.3.2,

1. we use PTEs to model the probabilistic behaviour of the system under observation, possibly starting from a HMM and then refining or extending it (Section 7.4.1);

2. we show how – by applying the rules defining the operational semantics

[1] http://www.runtime-verification.org/.

of PTEs – we obtain the same results of the forward algorithm presented in (Stoller et al., 2011) (Section 7.4.2);

3. by exploiting the algorithm presented in Section 4.5 and the Probabilize algorithm in cascade, we use PTEs also to model the linear time temporal properties that we want to verify at runtime;

4. by joining the two representations obtained in steps 1 and 3 above using the $\wedge$ conjunction operator natively provided by PTEs, we obtain for free a way to verify satisfaction of linear time temporal properties in presence of observation gaps (Section 7.4.3).

Considering the notion of Hidden Markov Model (HMM) presented in Section 3.8, we can now present a running example which will make the presentation of the chapter contents easier. We use the running example presented in (Stoller et al., 2011), where a planetary rover mission is modeled. The rover hosts two generic instruments, A and B, and all the events generated by the rover are recorded on a log file. Stoller and colleagues consider four different kinds of events, inspired by Barringer et al. (Barringer et al., 2010):

- *command* (`cmd` in the HMM figure), the command submitted to the rover;
- *dispatch* (`disp`), the dispatch of the command from the rover to the instrument;
- *success* (`succ`), the success of the command on the instrument;
- *fail* (`fail`), the failure of the command on the instrument.

All these events are characterized by three parameters: the instrument id (a or b), the issued command ($start$ or $reset$), and a time stamp indicating when the event occurred. When the rover receives a command, it reports the information to the logger and sends the command to the relevant instrument. Once received the command, the instrument issues a dispatch event to the logger and then executes the command. If the execution is successful (resp. fails), a corresponding success (resp. failure) event is reported to the logger. It is also possible that the command is simply lost for some reason and neither a success nor a fail occurs. All these events have some probability to be observed, and the chance to move from one state to another is also modeled by a probability. Figure 7.1 represents a HMM inspired to the rover example, where

- $S = \{s_1, s_2, s_3\}$;
- $A_{1,1} = A_{1,3} = 0; A_{1,2} = 1;$
  $A_{2,1} = 0.07; A_{2,2} = 0; A_{2,3} = 0.93;$
  $A_{3,1} = 1; A_{3,2} = A_{3,3} = 0;$
- $V = \{C, D, S, F\}$ (C stands for `cmd`, D for `disp`, etc);
- $b_1(C) = 1; b_1(D) = b_1(S) = b_1(F) = 0;$
  $b_2(D) = 1; b_2(C) = b_2(S) = b_2(F) = 0;$
  $b_3(C) = b_3(D) = 0; b_3(S) = 0.97; b_3(F) = 0.03;$

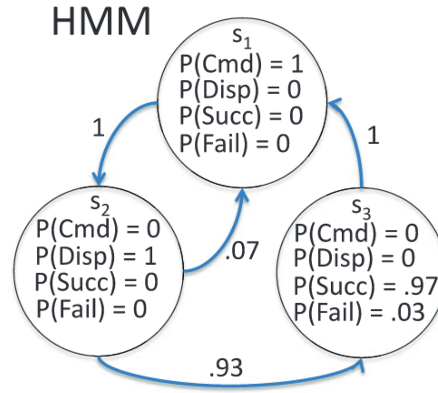

FIGURE 7.1. An example of HMM (from (Stoller et al., 2011)).

- $\pi_1 = 1, \pi_2 = \pi_3 = 0$ (not shown in the figure).

We can now introduce the concept of runtime verification with state estimation.

7.2 Runtime Verification with State Estimation

Given a trace (possibly with gaps), in (Stoller et al., 2011) Stoller et al. propose an approach to compute the probability that an LTL temporal property ϕ (Pnueli, 1977) is satisfied by a system modeled by a HMM H, given that $obs = O_1, O_2, ..., O_t$ have been observed. More formally, they evaluate $Pr(\phi|\ obs, H)$ by applying the following steps:

1. learn the HMM H from a given set of traces without gaps, using standard HMM learning algorithm;

2. generate the deterministic finite state machine (DFSM) corresponding to ϕ;

3. generate a monitor combining H and the DFSM to check the sequence obs.

Step 1 falls outside the boundaries of their investigation, and in the sequel we will disregard how the HMM has been created as well. For instance, a possible way to achieve this could be using the Python library *hmmlearn*[2] (Rabiner, 1990).

7.3 Probabilistic Trace Expressions

A probabilistic trace expression (PTE) is a trace expression where event types have a probability associated with them.

[2] https://hmmlearn.readthedocs.io/en/latest/

Example 3. *We present the PTE corresponding to the rover example introduced in Section 3.8. First of all we must define the event types. Considering the commands used in the model, we have the event type cmd = { command(Inst, Comm, TS) such that Inst ∈ {a, b}, Comm ∈ {start, reset}, TS a time stamp in the range 0...3 }, and in a similar way the event type disp = { dispatch(Inst, Comm, TS) }, succ = { success(Inst, Comm, TS) } and fail = { fail(Inst, Comm, TS) }. The resulting trace expression can be written in two equivalent (from the PTE semantics viewpoint) ways:*

$$\tau_{s_1} = cmd[1]{:}\tau_{s_2}$$
$$\tau_{s_2} = disp[0.07]{:}\tau_{s_1} \vee disp[0.93]{:}\tau_{s_3}$$
$$\tau_{s_3} = succ[0.97]{:}\tau_{s_1} \vee fail[0.03]{:}\tau_{s_1}$$

and

$$\tau'_{init} = cmd[1]{:}\tau'_{s_2}$$
$$\tau'_{s_1} = cmd[0.07]{:}\tau'_{s_2}$$
$$\tau'_{s_2} = disp[1]{:}(\tau'_{s_1} \vee \tau'_{s_3})$$
$$\tau'_{s_3} = succ[0.9021]{:}\tau'_{s_1} \vee fail[0.0279]{:}\tau'_{s_1}$$

The trace expression in the first form tells us, for example, that the probability of the protocol to reach τ_{s_1} starting from τ_{s_2} and having observed *disp* is 0.07 while the probability to reach τ_{s_3} starting from τ_{s_2} and having observed *disp* is 0.93 (second equation of the first formulation). To make this information explicit, the transition from state s_2 to states s_1 and s_3 in the HMM has been modeled by $\tau_{s_2} = disp[0.07]{:}\tau_{s_1} \vee disp[0.93]{:}\tau_{s_3}$, introducing non-determinism. While in a non probabilistic setting $\tau_{s_2} = disp{:}\tau_{s_1} \vee disp{:}\tau_{s_3}$ would be equivalent to $\tau_{s_2} = disp{:}(\tau_{s_1} \vee \tau_{s_3})$ and the second version would be definitely preferred, as – besides being more readable and compact – it is deterministic, in a probabilistic setting it would cause us to lose precious information on the probability to move in some state, given some observed event.

The second version overcomes this problem by propagating – via multiplication – the different probabilities associated with *disp* in s_2' to the states s_1' and s_3' that can be reached from s_2' (second and fourth equation of the second formulation). With this second form, we gain determinism at the price of adding an initial state τ_{init} for each state whose initial probability is not zero, and of losing the one-to-one clear correspondence with the HMM. As an example, in the fourth equation, understanding that $succ[0.9021]$ comes from the probability 0.97 associated with observing *succ* in state s_3' times the probability 0.93 of having reached s_3' from s_2' is not immediate.

Given that a structure-driven transformation from the first form to the second can be implemented in time linear with the trace expression length, we adopt the first form for presentation purposes, since it is closer to the HMM, but we use the second one in the implementation, since it is more efficient.

Like a "normal" trace expression, a PTE τ can be seen as the current state of a protocol that started in some initial state τ_{init} and reached τ after n events $O_1 \ldots O_n$ took place, that moved τ_{init} to τ through intermediate states τ_{q1}, τ_{q2}, ..., $\tau_{qn} = \tau$. If we denote with $\tau \xrightarrow{O} \tau'$ the transition from state τ to state τ' due to the event O taking place and being observed, we may write

$\tau_{init} \xrightarrow{O_1} \tau_{q1} \xrightarrow{O_2} \tau_{q2} \xrightarrow{O_3} \tau_{q3} \ldots \xrightarrow{O_n} \tau_{qn}$, where $\tau_{qn} = \tau$.

In order to properly manage probabilities, it is convenient to associate with τ – in an explicit and easily computable way – the probability of the protocol to have reached τ starting from τ_{init} and having observed $O_1 \ldots O_n$.

We define a "PTE state" (simply "state" from now on) the triple consisting of a trace expression τ, a sequence of events $O_1 \ldots O_n$ observed before reaching τ, and the probability π_τ that the protocol reached τ. We represent the state with the notation $\langle \tau, \pi_{tr}, O_1 \ldots O_n \rangle$.

In this work, we are interested in analyzing the protocol evolution in presence of observation gaps: the monitor driven by a PTE is aware that, in some state τ, some event took place and hence the protocol must move one step forward, but that event has not been correctly observed: the monitor perceived a "gap".

$$\text{(prefix)}\frac{}{\langle \vartheta[\pi_e]{:}\tau, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau, \pi_e * \pi_{tr}, obs\ any(e)\rangle}\ e \in \vartheta$$

$$\text{(or-l)}\frac{\langle \tau_1, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_1', \pi_{tr}', obs\ any(e)\rangle}{\langle \tau_1 \vee \tau_2, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_1', \pi_{tr}', obs\ any(e)\rangle}$$

$$\text{(or-r)}\frac{\langle \tau_2, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_2', \pi_{tr2}, obs\ any(e)\rangle}{\langle \tau_1 \vee \tau_2, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_2', \pi_{tr2}, obs\ any(e)\rangle}$$

$$\text{(and)}\frac{\langle \tau_1, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_1', \pi_{t1}, obs\ any(e)\rangle\ \langle \tau_2, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_2', \pi_{t2}, obs\ any(e)\rangle}{\langle \tau_1 \wedge \tau_2, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_1' \wedge \tau_2', \pi_{tr}', obs\ any(e)\rangle}\ \pi_{tr}' = f(\pi_{t1}, \pi_{t2})$$

$$\text{(shuffle-l)}\frac{\langle \tau_1, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_1', \pi_{tr}', obs\ any(e)\rangle}{\langle \tau_1 | \tau_2, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_1' | \tau_2, \pi_{tr}', obs\ any(e)\rangle}$$

$$\text{(shuffle-r)}\frac{\langle \tau_2, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_2', \pi_{tr}', obs\ any(e)\rangle}{\langle \tau_1 | \tau_2, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_1 | \tau_2', \pi_{tr}', obs\ any(e)\rangle}$$

$$\text{(cat-l)}\frac{\langle \tau_1, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_1', \pi_{tr}', obs\rangle}{\langle \tau_1 \cdot \tau_2, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_1' \cdot \tau_2, \pi_{tr}', obs\ any(e)\rangle}$$

$$\text{(cat-r)}\frac{\langle \tau_2, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_2', \pi_{tr}', obs\ any(e)\rangle}{\langle \tau_1 \cdot \tau_2, \pi_{tr}, obs\rangle \xrightarrow{any} \langle \tau_2', \pi_{tr}', obs\ any(e)\rangle}\ \varepsilon(\tau_1)$$

FIGURE 7.2. Transition system for probabilistic trace expressions states.

The transition rules between states are shown in Figure 7.2 and follow the

pattern of the rules defined for trace expressions, with modifications for taking care of the probability propagation and of observed events including gaps. The rules for ε are the same as for normal trace expressions.

In Figure 7.2 the use of *any* and $any(e)$ allows us to model the transition in the case that an event has been observed and in the case an observation gap took place, using the same rule. In fact, $any \in \{e, gap\}$ and if $any == e$ then $any(e) == e$; if $any == gap$ then $any(e) == gap(e)$.

If $any == e$, then e has been observed, the arrow modeling the state transition function $\stackrel{any}{\rightarrow}$ is actually labeled with e, and e is concatenated with the previously observed events, *obs*; if $any == gap$, then a gap took place, the arrow $\stackrel{any}{\rightarrow}$ is labeled with *gap*, and $gap(e)$, meaning that a gap took place, and that it could be filled with event e, is concatenated with the previously observed events. Differently from (Stoller et al., 2011), to perform runtime verification using PTEs, we need that each gap represents one single unobserved event: if we have a sequence of three unobserved events, we must have three different gaps in the observed trace. If, in the real system, this one event-one gap correspondence cannot be achieved, we can estimate the number of unobserved events that took place in a time slot T by computing the average rate of the event generation G, and inserting $T * G$ gaps in the event trace. As an example, if the monitor pauses for 3 seconds and the average events generation rate is 4 events for second, the trace will have 12 consecutive gaps corresponding to what happened in the time slot T.

More in details, the meaning of the transition rules is the following:

(prefix) If (1) the current state of the protocol is modeled by the trace expression $\tau_{current} = \vartheta[\pi_e]{:}\tau$, and (2) the probability to have reached $\tau_{current}$ after having observed *obs* events or gaps is π_{tr}, and (3.1) an event e having event type ϑ is observed, then the new protocol state is τ, with probability $\pi_e * \pi_{tr}$ (the probability to observe an event having type ϑ times the probability of $\tau_{current}$) and e is concatenated to the observed events, which become *obs e*. If in the third step a gap is observed, we have (3.2) instead: a gap is observed, which can be filled with any event $e \in \vartheta$, then the new protocol state is τ with probability $\pi_e * \pi_{tr}$, and $gap(e)$ is added to the observed events, which become *obs* $gap(e)$.

(or-l), (or-r) If, upon observation of *any*, either $\langle \tau_1, \pi_{tr}, obs\rangle$ or $\langle \tau_2, \pi_{tr}, obs\rangle$ can move into some state γ, then $\langle \tau_1 \vee \tau_2, \pi_{tr}, obs\rangle$ can move into γ. If both $\langle \tau_1, \pi_{tr}, obs\rangle$ and $\langle \tau_2, \pi_{tr}, obs\rangle$ can move, only one of them will do.

(and) If, upon observation of *any*, both $\langle \tau_1, \pi_{tr}, obs\rangle$ and $\langle \tau_2, \pi_{tr}, obs\rangle$ can move into $\langle \tau_1', \pi_{t1}, obs\ any(e)\rangle$ and $\langle \tau_2', \pi_{t2}, obs\ any(e)\rangle$ respectively, both branches of the $\wedge$ operator will move one step forward, reaching $\tau_1' \wedge \tau_2'$ through observation of *obs* $any(e)$. The probability of reaching this state is a function f of the probabilities associated with the states reached in the two branches, π_{t1} and π_{t2} respectively. It might be the average between π_{t1} and π_{t2}, the minimum between them if we want to adopt a cautious approach, the maximum if we adopt an optimistic approach. To

be as general as possible, we leave f unspecified in the presentation. We instantiate it with the minimum in the implementation of the transition system discussed in Section 7.5.

(shuffle-l), (shuffle-r) If, upon observation of *any*, $\langle \tau_1, \pi_{tr}, obs \rangle$ can move into $\langle \tau_1', \pi_{tr}', obs\ any(e) \rangle$, then the trace expression where τ_1 is the left branch of the | operator, and τ_2 is the right branch, can move into $\langle \tau_1' | \tau_2, \pi_{tr}', obs\ any(e) \rangle$ (the left branch "makes one step", the right one is kept). A similar rule holds when the right branch can move and the left one is kept. If both $\langle \tau_1, \pi_{tr}, obs \rangle$ and $\langle \tau_2, \pi_{tr}, obs \rangle$ can move, only one of them will do.

(cat-l) If, upon observation of *any*, $\langle \tau_1, \pi_{tr}, obs \rangle$ can move into $\langle \tau_1', \pi_{tr}', obs \rangle$, then the concatenation of τ_1 and τ_2, $\langle \tau_1 \cdot \tau_2, \pi_{tr}, obs \rangle$, can move into $\langle \tau_1' \cdot \tau_2, \pi_{tr}', obs\ any(e) \rangle$.

(cat-r) If τ_1 is ϵ or it "includes" ϵ and hence can "terminate" (for example, $\epsilon \vee \tau_3$) and, upon observation of *any*, τ_2 can move into τ_2' with probability π_{tr}', then the concatenation of τ_1 and τ_2, $\langle \tau_1 \cdot \tau_2, \pi_{tr}, obs \rangle$, can move into $\langle \tau_2', \pi_{tr}', obs\ any(e) \rangle$. Note that (cat-l) and (cat-r) are not mutually exclusive, as a trace expression can both "terminate" and evolve, like $\epsilon \vee \tau_3$. In this case, either (cat-l) or (cat-r) is applied.

7.3.1 Non Determinism in State Transitions

The state transition function $\stackrel{any}{\rightarrow}$ is non deterministic: one state can move into more than one state for many different reasons.

Let us consider the *cmd* event type introduced at the beginning of this section. The transitions below can take place starting from the state $\langle cmd[0.3]{:}\tau, 0.2, obs \rangle$ when an observation gap occurs.

- $\langle cmd[0.3]{:}\tau, 0.2, obs \rangle \stackrel{gap}{\rightarrow} \langle \tau, 0.06, obs\ gap(command(a, start, 0)) \rangle$
- $\langle cmd[0.3]{:}\tau, 0.2, obs \rangle \stackrel{gap}{\rightarrow} \langle \tau, 0.06, obs\ gap(command(b, start, 0)) \rangle$
- $\langle cmd[0.3]{:}\tau, 0.2, obs \rangle \stackrel{gap}{\rightarrow} \langle \tau, 0.06, obs\ gap(command(a, reset, 0)) \rangle$
- $\langle cmd[0.3]{:}\tau, 0.2, obs \rangle \stackrel{gap}{\rightarrow} \langle \tau, 0.06, obs\ gap(command(b, reset, 0)) \rangle$
- $\langle cmd[0.3]{:}\tau, 0.2, obs \rangle \stackrel{gap}{\rightarrow} \langle \tau, 0.06, obs\ gap(command(a, start, 1)) \rangle$
- ... plus 11 more transitions.

As another example, le us consider again the event type *cmd* defined above and the state

$$\langle cmd[0.75]{:}\tau_1 \vee cmd[0.25]{:}\tau_2, 0.4, obs \rangle$$

If $command(a, start, 3)$ (abbreviated in $c(a, s, 3)$ for presentation purposes) is observed, both branches of the choice in $cmd[0.75]{:}\tau_1 \vee cmd[0.25]{:}\tau_2$ are valid, leading to the two transitions below.

- $\langle cmd[0.75]{:}\tau_1 \vee cmd[0.25]{:}\tau_2, 0.4, obs\rangle \stackrel{c(a,s,3)}{\rightarrow} \langle \tau_1, 0.3, obs\ c(a,s,3)\rangle$
- $\langle cmd[0.75]{:}\tau_1 \vee cmd[0.25]{:}\tau_2, 0.4, obs\rangle \stackrel{c(a,s,3)}{\rightarrow} \langle \tau_2, 0.1, obs\ c(a,s,3)\rangle$

If, starting from $\langle cmd[0.75]{:}\tau_1 \vee cmd[0.25]{:}\tau_2, 0.4, obs\rangle$, a gap is observed, the two sources of nondeterminism (the first due to the gap that can be filled with many events matching the expected event type, and the second due to the nondeterministic choice in the trace expression) combine together, generating 32 possible transitions. Other sources of nondeterminism in the trace expression are due to the shuffle and the concatenation operators, defined by two transitions rules each.

$$\text{(state-to-set)}\frac{\gamma \stackrel{any}{\rightarrow} \gamma_1 \quad \gamma \stackrel{any}{\rightarrow} \gamma_2 \quad \dots \quad \gamma \stackrel{any}{\rightarrow} \gamma_n}{\gamma \stackrel{any}{\rightarrow}_\gamma \{\gamma_1, \gamma_2, \dots, \gamma_n\}}$$

$$\text{(set-to-set)}\frac{\gamma_1 \stackrel{any}{\rightarrow}_\gamma \Gamma_1 \quad \gamma_M \stackrel{any}{\rightarrow}_\gamma \Gamma_2 \quad \dots \quad \gamma_n \stackrel{any}{\rightarrow}_\gamma \Gamma_n}{\{\gamma_1, \gamma_2, \dots, \gamma_n\} \stackrel{any}{\twoheadrightarrow} \Gamma_1 \cup \Gamma_2 \cup \dots \cup \Gamma_1}$$

$$\text{(closure)}\frac{\Gamma_0 \stackrel{O_1}{\twoheadrightarrow} \Gamma_1 \stackrel{O_2}{\twoheadrightarrow} \quad \dots \quad \Gamma_{n-1} \stackrel{O_n}{\twoheadrightarrow} \Gamma_n}{\Gamma_0 \stackrel{O_1 \dots O_n}{\twoheadrightarrow} \Gamma_n}$$

$$\text{(closure-init)}\frac{\{\langle \tau, 1, \sigma\rangle\} \stackrel{O_1 \dots O_n}{\twoheadrightarrow} \Gamma_n}{\tau \stackrel{O_1 \dots O_n}{\twoheadrightarrow} \Gamma_n} \ \sigma\text{=empty sequence}$$

FIGURE 7.3. Rules for nondeterminism and transitive closure.

Figure 7.3 presents the rules for dealing with non determinism and for introducing the notion of transitive closure of transitions:

(state-to-set) The function represented by $\rightarrow_\gamma$ takes one PTE state γ, one observed event or gap *any*, and returns the set of all the PTE states that γ can reach via $\stackrel{any}{\rightarrow}$.

(set-to-set) The function represented by $\twoheadrightarrow$ takes one set of PTE states $\{\gamma_1, \gamma_2, \dots, \gamma_n\}$, one observed event or gap *any*, and returns the union of the sets of PTE states that each $\gamma_i \in \{\gamma_1, \gamma_2, \dots, \gamma_n\}$ can reach via $\stackrel{any}{\rightarrow}_\gamma$.

(closure) We use $\twoheadrightarrow$ to denote the transitive closure of $\twoheadrightarrow$ by putting the sequence of observed events on top of the arrow.

(closure-init) Finally, a PTE τ can evolve into any state $\gamma \in \Gamma_n$ after observation of $O_1 \dots O_n$, if the PTE state $\langle \tau, 1, \sigma\rangle$ can.

Example 4. *Starting from the PTE τ_{s_1} used as running example, we have:*

$$\tau_{s_1} \stackrel{cmd\ disp\ gap}{\twoheadrightarrow}$$

$$\{\langle \tau_{s_2}, 0.07, cmd\ disp\ gap(cmd)\rangle,$$

$$\langle \tau_{s_1}, 0.9021, cmd\ disp\ gap(succ)\rangle,$$

$$\langle \tau_{s_1}, 0.0279, cmd\ disp\ gap(fail)\rangle\}$$

because

$$\langle \tau_{s_1}, 1, []\rangle \xrightarrow{cmd} \{\langle \tau_{s_2}, 1, cmd\rangle\} \overset{disp}{\twoheadrightarrow} \{\langle \tau_{s_1}, 0.07, cmd\ disp\rangle, \langle \tau_{s_3}, 0.93, cmd\ disp\rangle\} \overset{gap}{\twoheadrightarrow}$$
$$\{\langle \tau_{s_2}, 0.07, cmd\ disp\ gap(cmd)\rangle,$$
$$\langle \tau_{s_1}, 0.9021, cmd\ disp\ gap(succ)\rangle,$$
$$\langle \tau_{s_1}, 0.0279, cmd\ disp\ gap(fail)\rangle\}$$

7.3.2 From Trace Expressions to Probabilistic Trace Expressions

Since PTEs are an extension of standard trace expressions, we need a "Probabilize" algorithm to convert a standard trace expression into a probabilistic one. This allows us to combine "probabilized" standard trace expression together with probabilistic ones, opening important possibilities as discussed in Section 7.4.3 and ensuring backward compatibility.

In this section we provide a sketch for the "Probabilize" algorithm, that is under implementation. Given a non probabilistic trace expression τ_{np}, we can obtain its corresponding probabilistic version $Probabilize(\tau_{np})$ by adding probabilities parameters to all the event types that appear in τ_{np}. To achieve this result, we have to define an algorithm that operates on τ_{np} following its structure and that, when there are more than one possible moves from the current state to the next ones due to observability of different event types, shares the probability among these event types following some probability distribution, the uniform one in the simplest case. For instance, if the algorithm is currently analyzing the state $cmd : \tau_{s_1} \vee disp : \tau_{s_2}$ and if it is using a uniform distribution probability, it must extract the set of event types that can lead to a new state $\{cmd, disp\}$, count them obtaining a number n (in this case $n = 2$), and assign probability $1/n$ to each of them. In this case the resulting probability is 0.5 and

$$Probabilize(cmd : \tau_{s_1} \vee disp : \tau_{s_2}) = cmd[0.5] : \tau_{s_1} \vee disp[0.5] : \tau_{s_2}$$

The shuffle operator, |, is the only operator maintaining information on the past (when we consume a branch we preserve the other one in the new state). It must be handled in a different, and more careful, way, that we are currently working out.

Since trace expressions can be cyclic terms, the *Probabilize* algorithm must take care of cycles. This issue can be addressed by exploiting coinduction. As an example, the use of the SWI-Prolog coinduction library allows to cope with infinite terms without entering into loops, and given that we model trace expressions as Prolog terms, we will take advantage of this feature as we did in the rest of the thesis.

7.4 From Hidden Markov Models to Probabilistic Trace Expressions

Writing from scratch a PTE where probabilities associated with event types are consistent with their intended meaning and with the probability properties is not an easy task. Besides needing a deep knowledge of modeled system, the developer would also need a means to ensure that, for example, a PTE like $cmd[0.9] : \tau_{s_1} \vee disp[0.8] : \tau_{s_2}$ is recognized as wrong, since there are two mutually exclusive branches and the sum of their probabilities is greater than one. While this error is trivial and can be easily catched and corrected, if the PTE grows in size and complexity a manual development becomes more and more error-prone.

A good practice in engineering new software applications is to reuse well established approaches as much as possible. Even if we want to model probabilistic systems using an extension of trace expressions, which is more expressive than HMM and deterministic finite state machines, this does not prevent us from starting from a less expressive but widely used formalism like HMM in order to extract useful information, and refine it if necessary.

If a HMM representing the behaviour of the modeled system exists, for example because it has been learned using existing algorithms, we can use it to generate the corresponding PTE in an automatic way. Once such PTE has been obtained, we can modify it in order to model those features of the actual system that could not be directly represented with a HMM.

7.4.1 The HMM2PTE Algorithm

Given a HMM $H = \langle S, A, V, B, \Pi \rangle$, the algorithm to construct an equivalent PTE is the following:

1. for each observation symbol $v_k \in V$, generate the corresponding singleton event type $\beta_k = \{v_k\}$ (recall that trace expressions are defined on top of event types and not of events);
2. for each $i = 1..N_s$, for each $j = 1..N_s$, for each $k = 1..N_v$, if $A_{i,j} \neq 0$ then $\tau_{s_i} = \bigvee_{j=1..N_s, k=1..N_v} \beta_k[A_{i,j} * b_i(v_k)]{:}\tau_{s_j}$[3]. If, for some given i, there exists only one j such that $A_{i,j}$ is different from 0, then $\tau_{s_i} = \beta_k[A_{i,j} * b_{i,k}]{:}\tau_{s_j}$. If, for some given i, all $A_{i,j}$ are equal to 0, then $\tau_{s_i} = \epsilon$.

As an example, the HMM2PTE algorithm translates the HMM presented in Section 3.8 into the PTE presented in Section 7.3.

7.4.2 Forward Algorithm for Probabilistic Trace Expressions

Let us consider the set of PTEs states

$$\Gamma_0 = \{\langle \tau_{s_1}, \pi_{s_1}, \sigma \rangle, \langle \tau_{s_2}, \pi_{s_2}, \sigma \rangle, ..., \langle \tau_{s_N}, \pi_{s_N}, \sigma \rangle\}$$

[3] By $\bigvee_{h=1..m} \tau_h$ we mean the conjunction via the $\vee$ operator of the trace expressions $\tau_1, ..., \tau_m$. The notation can only be used if $m \geq 2$.

where each τ_{s_i} corresponds to a state s_i in the HMM H and has been obtained applying the translation algorithm. π_{s_1} is the initial probability of s_1, according to H. If $\pi_{s_1} = 0$, the corresponding state is discarded.

We remind that – given a correct HMM – the sum of the elements $A_{i,j}$ of the ith row of A must be one, as $A_{i,j}$ represents the probability to reach state s_j at time $t+1$ starting from state s_i at time t, and the total probability to move to some next state must be 1: $\Sigma_{j=1}^{N_s} A_{i,j} = 1$ for each i.

We also remind that $\alpha_t(i) = Pr(O_1, O_2, ..., O_t, q_t = s_i|H)$, i.e., the probability that the first t observations yield $O_1, O_2, ..., O_t$ and that q_t is s_i, given the model H. The definition of α according to the forward algorithm presented in (Stoller et al., 2011) (see Section 3.8 for further information) is the following.
Base case:

$$\alpha_1(j) = \pi_j b_j(O_1) \text{ for } 1 \leq j \leq N_s$$

Recursive case:

$$\alpha_{t+1}(j) = (\Sigma_{i=1..N_s} \alpha_t(i) A_{i,j}) b_j(O_{t+1}) \text{ for } 1 \leq t \leq T-1 \text{ and } 1 \leq j \leq N_s$$

If $\Gamma_0 \overset{O_1...O_{t-1}}{\twoheadrightarrow} \Gamma_{t-1}$, we can consider all the states in Γ_{t-1} of the form $\langle \tau_{s_i}, \pi, O_1...O_{n-1} \rangle$, namely those states whose trace expression is τ_{s_i}. We denote with $\Gamma_{t-1}(\tau_{s_t}) = \{\langle \tau_{s_t}, \pi, O_1...O_{t-1} \rangle \in \Gamma_{t-1}\}$.

Theorem 2. *If* $\Gamma_0 \overset{O_1...O_t}{\twoheadrightarrow} \Gamma_t$*, then* $\alpha_t(j) = \Sigma_{\langle \tau_{s_j}, \pi_j, O_1...O_t \rangle \in \Gamma_t} \pi_j$ *(for* $1 \leq j \leq N_s$*).*

First, we give the intuition behind the theorem by means of our running example, then we propose a proof sketch.

Let us suppose that we want to compute the probability that, after observing `command(a, start, 0)` (C in the sequel), `dispatch(a, start, 1)` (D in the sequel), `fail(a, start, 2)` (F in the sequel), the system is in state s_3.

We have to compute $\Gamma_0 \overset{O_1...O_{t-1}}{\twoheadrightarrow} \Gamma_{t-1}$ first, namely $\Gamma_0 \overset{CD}{\twoheadrightarrow} \Gamma_{t-1}$.

$$\Gamma_0 = \{\langle \tau_{s_1}, 1, \sigma \rangle\} \overset{C}{\twoheadrightarrow} \{\langle \tau_{s_2}, 1, C \rangle\} \overset{D}{\twoheadrightarrow} \{\langle \tau_{s_1}, 0.07, CD \rangle, \langle \tau_{s_3}, 0.93, CD \rangle\}$$

Once reached $\{\langle \tau_{s_1}, 0.07, CD \rangle, \langle \tau_{s_3}, 0.93, CD \rangle\}$ we have to limit the last transition, tagged with F, to those states whose trace expression corresponds to s_3, namely τ_{s_3}.

$$\{\langle \tau_{s_1}, 0.07, CD \rangle, \langle \tau_{s_3}, 0.93, CD \rangle\}(\tau_{s_3}) = \{\langle \tau_{s_3}, 0.93, CD \rangle\} \overset{F}{\twoheadrightarrow} \{\langle \tau_{s_1}, 0.0279, CDF \rangle\}$$

It turns out that $\alpha_{CDF}(3)$ = probability to observe CDF, with F observed in state s_3, should be 0.0279. We use CDF as subscript for α for sake of clarity. Let us compute the same value using the forward algorithm adapted for PTEs. The base case leads to the following computation:

$$\alpha_C(1) = \pi_1 * b_1(C) = 1 * 1 = 1$$

$$\alpha_C(2) = \pi_2 * b_2(C) = 0 * 0 = 0$$

$$\alpha_C(3) = \pi_3 * b_3(C) = 0 * 0 = 0$$

The first recursive step leads to the following computation (we omit some details and keep the result):

$$\alpha_{CD}(1) = (\Sigma_{i=1..N_s} \alpha_C(i) A_{i,1}) b_1(D) = 0$$

$$\alpha_{CD}(2) = (\Sigma_{i=1..N_s}\alpha_C(i)A_{i,2})b_2(D) = 1 * A_{1,2} * b_2(D) = 1 * 1 * 1 = 1$$
$$\alpha_{CD}(3) = (\Sigma_{i=1..N_s}\alpha_C(i)A_{i,3})b_3(D) = 0$$

The third recursive step leads to:

$$\alpha_{CDF}(1) = (\Sigma_{i=1..N_s}\alpha_{CD}(i)A_{i,1})b_1(F) = 0$$
$$\alpha_{CDF}(2) = (\Sigma_{i=1..N_s}\alpha_{CD}(i)A_{i,2})b_2(F) = 0$$
$$\alpha_{CDF}(3) = (\Sigma_{i=1..N_s}\alpha_{CD}(i)A_{i,3})b_3(F) = \alpha_{CD}(2) * A_{2,3} * b_3(F) = 1 * .97 * .03 = .0279$$

Let us consider the set of PTEs states $\Gamma_0 = \{\langle\tau_{s_1}, \pi_{s_1}, \sigma\rangle, \langle\tau_{s_2}, \pi_{s_2}, \sigma\rangle, ..., \langle\tau_{s_N}, \pi_{s_N}, \sigma\rangle\}$, where each τ_{s_i} corresponds to a state s_i in the HMM H and has been obtained applying the translation algorithm. π_{s_1} is the initial probability of s_1, according to H. If $\pi_{s_1} = 0$, the corresponding state is discarded.

Proof Sketch. By induction on t. In the demonstration we fix the final state $q_t = s_i$ and we name it s_{last} to avoid confusion w.r.t. the indexes of the observed events and the state identifiers in the HMM and in the PTEs.

Base case. According to the forward algorithm, when $t = 1$, $\alpha_1(last) = \pi_{last} * b_{last}(O_1)$.
According to our theorem,
if $\Gamma_0(\tau_{s_{last}}) \overset{O_1}{\twoheadrightarrow} \Gamma_1$,
then $\alpha_1(last) = \Sigma_{\langle\tau_{s_{last}}, \pi_{last}, O_1\rangle \in \Gamma_1}\pi_{last}$
namely, $\alpha_1(last)$ is the probability that starting from $\tau_{s_{last}}$ some state has been reached (we are not interested in knowing which one) after observing O_1. If more than one state can be reached from $\tau_{s_{last}}$ after observing O_1, we sum the probabilities associated with those states.

$\Gamma_0(\tau_{s_{last}}) = \{\langle\tau_{s_{last}}, \pi_{s_{last}}, \sigma\rangle\}$ where $\pi_{s_{last}}$ is the initial probability of s_{last} according to the HMM H.

Let us define $\beta_1 = \{O_1\}$. By construction, $\tau_{s_{last}} = \bigvee_{j=1..N_s, k=1..N_v} \beta_k[A_{last,j} * b_{last}(v_k)]{:}\tau_{s_j}$ and hence, after observation of O_1, only those or-branches whose event type is β_1 and that lead to some state τ_{s_j} with probability $A_{last,j} * b_{last}(O_1)$ are kept.

Each of these branches generates one state $\langle\tau_{s_j}, \pi_{s_{last}} * A_{last,j} * b_{last}(O_1), O_1\rangle \in \Gamma_1$ because $\pi_{s_{last}}$ associated with $\tau_{s_{last}}$ in Γ_0 is multiplied – for the semantics of the prefix operator in PTEs, rule "prefix" in Figure 7.2 – with $A_{last,j} * b_{last}(O_1)$ which is the probability associated with β_1 in $\beta_1[A_{last,j} * b_1(O_1)]{:}\tau_{s_j}$.

Given that $\Sigma_{j=1}^{N_s} A_{last,j} = 1$, by summing $\pi_{s_{last}} * A_{last,j} * b_{last}(O_1)$ for all states s_j that can be reached by observing O_1 in s_{last} we obtain $\Sigma_{j=1}^{N_s}(\pi_{s_{last}} * A_{last,j} * b_{last}(O_1)) = (\pi_{s_{last}} * b_{last}(O_1)) * (\Sigma_{j=1}^{N_s} A_{last,j}) = \pi_{s_{last}} * b_{last}(O_1) * 1$.

This demonstrates that $\Sigma_{\langle\tau_{s_j}, \pi_j, O_1\rangle \in \Gamma_1}\pi_j$ has the same value as $\alpha_1(last) = \pi_{last} * b_{last}(O_1)$ according to the forward algorithm.

Inductive step.

We assume that

$$\alpha_t(j) =$$
$$(\Sigma_{i=1..N_s}\alpha_{t-1}(i)A_{i,j})b_j(O_t) \text{ for } 1 \le t \le T-2 \text{ and } 1 \le j \le N_s =$$
$$\Sigma_{\langle \tau_{s_j}, \pi_j, O_1...O_t\rangle \in \Gamma_t}\pi_j \text{ (for } t \ge 0 \text{ and } 1 \le j \le N_s)$$

We have to demonstrate that

$$\alpha_{t+1}(j) =$$
$$(\Sigma_{i=1..N_s}\alpha_t(i)A_{i,j})b_j(O_{t+1}) \text{ for } 1 \le t \le T-1 \text{ and } 1 \le j \le N_s =$$
$$\Sigma_{\langle \tau_{s_j}, \pi_j, O_1...O_{t+1}\rangle \in \Gamma_{t+1}}\pi_j \text{ (for } t \ge 0 \text{ and } 1 \le j \le N_s)$$

Also in this case we fix the final state we expect to reach, s_{last} ($last$ is the $t+1$th state), and we demonstrate that $\alpha_{t+1}(last) = \Sigma_{\langle \tau_{s_{last}}, \pi_{last}, O_1...O_{t+1}\rangle \in \Gamma_{t+1}}\pi_{last}$.

From the theorem hypothesis, we have that $\Gamma_0 \overset{O_1...O_{t+1}}{\twoheadrightarrow} \Gamma_{t+1}$. Since we denote the states belonging to Γ_t as $\tau_{s_{last}}$, we can also say that $\Gamma_t(\tau_{s_{last}}) \overset{O_{t+1}}{\twoheadrightarrow} \Gamma_{t+1}$.

Let us suppose that s_{last} in Γ_t can be reached from $s_{last-1}, ..., s_{last-k}$ in Γ_{t-1} after observing O_t.

This means that Γ_{t-1} contains the following states

$$\langle \tau_{s_{last-1}}, \pi_{last-1}, O_1...O_{t-1}\rangle$$
$$\langle \tau_{s_{last-2}}, \pi_{last-2}, O_1...O_{t-1}\rangle$$
$$...$$
$$\langle \tau_{s_{last-k}}, \pi_{last-k}, O_1...O_{t-1}\rangle$$

From the way we build the transition $\Gamma_{t-1} \overset{O_t}{\twoheadrightarrow} \Gamma_t$, $\Gamma_t(\tau_{s_{last}})$ contains the following states

$$\langle \tau_{s_{last}}, \pi_{last-1} * A_{last-1,last} * b_{last-1}(O_t), O_1...O_t\rangle$$
$$\langle \tau_{s_{last}}, \pi_{last-2} * A_{last-2,last} * b_{last-2}(O_t), O_1...O_t\rangle$$
$$...$$
$$\langle \tau_{s_{last}}, \pi_{last-k} * A_{last-k,last} * b_{last-k}(O_t), O_1...O_t\rangle$$

and from the way we build the transition $\Gamma_t(\tau_{s_{last}}) \overset{O_{t+1}}{\twoheadrightarrow} \Gamma_{t+1}$, if we suppose that observing O_{t+1} in $\tau_{s_{last}}$ leads to $\tau_{s_{last+1}}$, ..., $\tau_{s_{last+h}}$, Γ_{t+1} contains the following states

$$\langle \tau_{s_{last+1}}, \pi_{last-1} * A_{last-1,last} * b_{last-1}(O_t) * A_{last,last+1} * b_{last}(O_{t+1}), O_1...O_{t+1}\rangle$$
$$\langle \tau_{s_{last+1}}, \pi_{last-2} * A_{last-2,last} * b_{last-2}(O_t) * A_{last,last+1} * b_{last}(O_{t+1}), O_1...O_{t+1}\rangle$$
$$...$$
$$\langle \tau_{s_{last+1}}, \pi_{last-k} * A_{last-k,last} * b_{last-k}(O_t) * A_{last,last+1} * b_{last}(O_{t+1}), O_1...O_{t+1}\rangle$$

$$\langle \tau_{s_{last+2}}, \pi_{last-1} * A_{last-1,last} * b_{last-1}(O_t) * A_{last,last+2} * b_{last}(O_{t+1}), O_1...O_{t+1}\rangle$$
$$\langle \tau_{s_{last+2}}, \pi_{last-2} * A_{last-2,last} * b_{last-2}(O_t) * A_{last,last+2} * b_{last}(O_{t+1}), O_1...O_{t+1}\rangle$$

$$\dots$$

$$\langle \tau_{s_{last+2}}, \pi_{last-k} * A_{last-k,last} * b_{last-k}(O_t) * A_{last,last+2} * b_{last}(O_{t+1}), O_1 \dots O_{t+1} \rangle$$

$$\dots$$

$$\langle \tau_{s_{last+h}}, \pi_{last-1} * A_{last-1,last} * b_{last-1}(O_t) * A_{last,last+k} * b_{last}(O_{t+1}), O_1 \dots O_{t+1} \rangle$$

$$\langle \tau_{s_{last+h}}, \pi_{last-2} * A_{last-2,last} * b_{last-2}(O_t) * A_{last,last+h} * b_{last}(O_{t+1}), O_1 \dots O_{t+1} \rangle$$

$$\dots$$

$$\langle \tau_{s_{last+h}}, \pi_{last-k} * A_{last-k,last} * b_{last-k}(O_t) * A_{last,last+h} * b_{last}(O_{t+1}), O_1 \dots O_{t+1} \rangle$$

The probabilities of the states in Γ_{t+1} that can be reached from $\tau_{s_{last}}$ by observing O_{t+1} have the form $\pi_{last-i} * A_{last-i,last} * b_{last-i}(O_t) * A_{last,last+j} * b_{last}(O_{t+1})$ with $i = 1..k,\ j = 1..h$.

We can express the formula as

$$b_{last}(O_{t+1}) * \Sigma_{j=1}^{h}(A_{last,last+j} * \Sigma_{i=1}^{k}(\pi_{last-i} * A_{last-i,last} * b_{last-i}(O_t)))$$

and we must demonstrate that this formula is equal to

$$\alpha_{t+1}(last) =$$

$$(\Sigma_{i=1}^{k} \alpha_t(last - i) A_{last-i,last}) b_{last}(O_{t+1})$$

The formula holds if

$$\Sigma_{i=1}^{k}(\pi_{last-i} * A_{last-i,last} * b_{last-i}(O_t))$$

is equal to

$$\Sigma_{i=1}^{k} \alpha_t(last - i) A_{last-i,last}$$

as the external sum $\Sigma_{j=1}^{h} A_{last,last+j}$ gives 1, since it is the total probability that a move is made from τ_{last} to some other state.

The equation

$$\alpha_t(last - i) = (\pi_{last-i} * b_{last-i}(O_t))$$

holds if

$$\pi_{last-i} = \Sigma_{i=1..N_s} \alpha_t(i) A_{i,last-i}$$

and, finally, this last equation holds from the way we propagate probabilities from global states to global states.

7.4.3 Satisfying LTL Properties when Gaps Are Observed

Satisfaction of LTL properties in presence of observation gaps can be verified in a natural and elegant way thanks to

- the possibility to represent an LTL property as a standard trace expression (Section 4.5),

- the possibility to transform a standard trace expression into a probabilistic one thanks to the *Probabilize* algorithm introduced in Section 7.3.2, and
- the "and" operator, $\wedge$, modeling the fact that the (probabilistic) trace expressions in the two branches must perform the same transitions. From an set-theoretic viewpoint, $\wedge$ models the intersection of the event traces represented by the two branches it joins.

Let us identify with τ_{HMM} the PTE representing a HMM, and with τ_ϕ the standard trace expression representing the temporal property ϕ to be verified.

The PTE $\tau_{HMM} \wedge Probabilize(\tau_\phi)$ models the intersection of the languages recognized by the HMM and ϕ: by making the intersection of the states in τ_{HMM} with those in $Probabilize(\tau_\phi)$ we automatically constrain the evaluation process to those traces produced by the HMM that respect ϕ.

7.5 Implementation and Experiments

We implemented all the algorithms presented in this chapter, apart from the "Probabilize" one which is under development, using SWI-Prolog. Since we are interested in applying the *Probabilize* algorithm to a trace expression equivalent to a LTL property ϕ, it is enough using a simplified version of the algorithm, easier to implement, because the trace expression τ_ϕ does not contain the shuffle operator by construction (Chapter 4). The corresponding translation thus can be achieved as presented in Section 7.3.2, because we have not to consider any special cases.

PTEs can be implemented in SWI-Prolog exactly as normal trace expressions (Section 14.2).

Thanks to Prolog's rule-based, declarative interpretation, the rules defining trace expressions operational semantics have a one-to-one correspondence with Prolog clauses: backtracking and "all-solutions" predicates are powerful tools to deal with the generation of multiple PTE states, when gaps introduce non determinism (*set-to-set* rule). Finally, as we already anticipated, the SWI-Prolog coinduction library allows us to manipulate cyclic terms avoiding loops in the execution.

Events can be observed as an online stream while they are generated by the system (*online RV*), or can be recorded on a log file and then inspected (*offline RV*). In both scenarios there may be gaps, due to different reasons. In offline RV, gaps might be caused by event sampling, as usually done to reduce the monitor workload. In online RV, a gap indicates lack of information (a lost message, event or perception); in this case, the absence of information is related to technical constraints of the system or of the monitor observation capabilities rather than to optimization purposes.

The *set-to-set* semantic rule generates a set of states each time it is applied. Even though for normal trace expressions it is enough that SWI-Prolog maintains the only one current state (Section 14.2), for PTEs SWI-Prolog needs to maintain the set of current states in its local knowledge base (as we have seen this is due to the presence of gaps).

Unfortunately, a rule like *set-to-set* suffers from state space explosion, in particular when there are many sources of non determinism. Each time a gap takes place, the monitor must make guesses on the possible actual events that the gap represents and save all the states generated by these guesses. A possibly huge logical tree-like structure with states as nodes, and moves from states to states as edges, represents these open possibilities. If the RV takes place online, the exploration of this logical structure should follow a breadth-first strategy (more space needed but possibly less time required to recognize that the trace is not compliant with the expected behaviour), as the final trace of events is unknown and the levels of the structure are generated and explored at the same time. When, instead, a log file is analyzed offline, the trace in the log is already complete and the logical tree-like structure can be explored, looking for violations, following a depth-first search (less space needed, but the violation could be discovered after exploring all the structure).

7.6 Discussion

In this chapter we addressed the presence of gaps in observed traces and the need to estimate the probability that the (incomplete) traces satisfy some LTL properties, when the system is modelled by a PTE. Although PTEs have a higher potential expressive power than HMM and LTL, being able to express traces like $a^n b^n c^n$, in this work we start from a HMM of the real system and generate an equivalent PTE from it, which of course is as expressive as the HMM it originates from. This is a safe approach to generate a PTE consistent with the known probability distribution of events, which can then be refined in such a way that its expressiveness is fully exploited. In Chapter 10 we are going to present some more examples of possible uses of this probabilistic extension.

Part IV

Engineering Agent Interaction Protocols

This part of the thesis focuses on the use of the trace expression formalism to represent Agent Interaction Protocols (AIP). Chapter 8 focuses on the issues we can encounter when we define and verify agent interaction protocols. Chapter 9 and 10 present and solve the problem of how to decentralize the runtime verification of an agent interaction protocol without and with uncertainty in the messages, respectively. Finally, Chapter 11 puts the basis on how two trace expressions representing agent interaction protocols can be compared. This is obtained through the definition of a conformance relation among different specifications on different domains.

8 Issues with Agent Interaction Protocols

"Much unhappiness has come into the world
because of bewilderment and things left unsaid."
- Fyodor Dostoyevsky

Interaction Protocols are fundamental elements to provide the entities in a system, be them actors, agents, services, or other communicating pieces of software, a means to agree on a global interaction pattern and to be sure that all the other entities in the system adhere to it as well. Tagging some protocols as good ones and others as bad is common to all the research communities where interaction is crucial, and it is not surprising that some protocol features are recognized as bad ones everywhere. In this chapter we analyze the notion of good, bad and ugly protocols in the MAS community and outside, and we explore the possibility that bad protocols are not that bad, after all. In particular, we concentrate on the problem of MAS monitoring under the assumption of partial observability*: even if the global protocol ruling the MAS is a good one, partial or no observability of some events therein can make the actually monitorable protocol a bad one, with covert channels due to unobservable events. An observability-driven protocol transformation algorithm is presented, and its implementation and experiments with the trace expression formalism are discussed.*

The contents of this chapter are published in
(Ancona et al., 2018a)

8.1 Introduction

Interaction Protocols are a key ingredient in MAS as they explicitly represent the agents expected/allowed communicative patterns and can be used either to check the compliance of the agent actual behavior w.r.t. expected one (Alberti et al., 2005; Baldoni et al., 2005a) or to drive the agent behavior itself (Ancona et al., 2015a).

Interaction protocols are also crucial outside the MAS community: what we name an "Agent Interaction Protocol", is referenced as a "Choreography" in the Service Oriented Computing (SOC) community (Papazoglou, 2003) and as a "Global Type" in the multiparty session types one (Bettini et al., 2008; Honda, Yoshida, and Carbone, 2008).

In the MAS community, AIPs describe interaction patterns characterizing the system as a whole. This global viewpoint is supported by many formalisms and notations such as AUML (Huget and Odell, 2004), commitment machines and their extensions, the Blindingly Simple Protocol Language (BSPL) and its Splee extension, the Hierarchical Agent Protocol Notation (HAPN)[1], and of course trace expressions (Chapter 4).

When moving from the specification to the execution stage, the AIP must be enacted by agents in the MAS: besides the global description of the protocol, the "local" description of the AIP portion each agent is in charge of, is required to run the AIP. The AIP enactment is usually left to Computer-Aided Software Engineering tools that move from AIP diagrams directly into agent skeletons in some concrete agent oriented programming language (Cossentino, 2005; García-Ojeda, DeLoach, and Robby, 2009; Padgham and Winikoff, 2002), or to algorithms that translate the AIP textual representation to some abstract, intermediate formalism for modeling the local viewpoint (Casella and Mascardi, 2007; Desai et al., 2005b). Such intermediate formalisms are not perceived as the main target of the research and no standardization effort has been put on them.

In the SOC community, on the contrary, formalisms exist for modeling both the global and the local perspectives. As observed by (Lanese et al., 2008), WS-CDL[2] follows an interaction-oriented ("global") approach, whereas in BPEL4Chor[3] the business process of each partner involved in a choreography is specified using an abstract version of BPEL[4]: BPEL4Chor follows a process-oriented ("local") approach.

In the multiparty session types community, the main emphasis is on type-checking aspects: the formalism used to represent global types is relevant, as well as its expressive power, but even more relevant are the properties of the "global to local" projection function w.r.t. type-safety issues (Deniélou and

[1]See Chapter 5 for further information on these formalisms.

[2]Web Services Choreography Description Language Version 1.0 W3C Candidate Recommendation 9 November 2005, `https://www.w3.org/TR/ws-cdl-10/`.

[3]BPEL4Chor Choreography Extension for BPEL, `http://www.bpel4chor.org/`.

[4]Web Services Business Process Execution Language Version 2.0, OASIS Standard, 11 April 2007, `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

Yoshida, 2012).

Whatever the research area, assumptions on the protocols are made which allow them to be classified as good, bad and ugly. Whereas the classification of ugly protocols may depend on the protocol purpose and on the formalism used to express it, almost all the authors agree on tagging as "bad" the same classes of protocols based on problems they might raise during the projection and/or enactment stages.

Before going on with the presentation of what is a "good", a "bad" and a "ugly" protocol, we need to re-introduce briefly the concept of projection of an AIP, and we need to comment a little bit how our work is related with the state of the art.

8.2 Projection of an Agent Interaction Protocol

With an AIP, we can represent the behaviour of each agent inside the system globally. When AIPs are used for runtime verification purposes, since we have only one monitor checking the entire system, it is reasonable that all the information collapses in the protocol representation (which encodes what all agents can or cannot do). However, when we are interested in verifying at runtime the compliance of the AIP with respect to an existent MAS, it can be necessary to split up the workload among multiple monitors (trying to avoid the bottleneck problem concerning the use of a single centralized one).

Starting from an AIP and an agent set, we can define a *project* function Π as the function returning the AIP's local view with respect to an agent set (Ancona et al., 2014, 2016). If we take, for instance, an AIP whose interaction events involve the set of agents Ags = {agent_1, agent_2, agent_3}. We can use the *project* function passing the protocol and the subset {agent_1, agent_3} in order to obtain a local view of the protocol representing only the event traces involving at least agent_1 or agent_3. It is important to note that the event traces represented by the local AIP (LAIP) do not involve only the agents passed to the *project* function. For example, if we have an event involving agent_1 and agent_2, it would still be present in the projected AIP (since agent_1 is involved).

The *project* function can be defined in the following way:

$$\Pi\colon AIP \times \mathcal{P}(Ags) \to AIP$$

where the second argument is the subset of agents onto which projection is made.

The *project* function allows us to distribute the protocol among more monitors instead of only one.

In the specific scenario of trace expressions, the projection function can be rewritten as

$$\Pi : \mathcal{T} \times \mathcal{P}(Ags) \to \mathcal{T}$$

where $\mathcal{T}$ is the set of trace expressions.

For what concerns the contents of the thesis, the given high-level presentation is enough to understand the contributions; but, for further details, the

formal definition and computation of the projection function can be found in (Ancona et al., 2014, 2016).

8.3 State of the art

To the best of our knowledge, MAS monitoring under partial or imperfect observability has been addressed in the context of normative multi-agent organizations only, and by just a few works. Among them, (Alechina, Dastani, and Logan, 2014) spun off from (Bulling, Dastani, and Knobbout, 2013) and shows how to move from the heaven of ideal norms to the earthly condition of approximate norms. The chapter focuses on conditional norms with deadlines and sanctions (Tinnemeier et al., 2009); ideal norms are those that can be perfectly monitored given a monitor, and optimality of a norm approximation means that any other approximation would fail to detect at least as many violations of the ideal norm. Given a set of ideal norms, a set of observable properties, and some relationships between observable properties and norms, the chapter presents algorithms to automatically synthesize optimal approximations of the ideal norms defined in terms of the observable properties. Even if the purpose of our work is in principle similar to that of (Alechina, Dastani, and Logan, 2014; Bulling, Dastani, and Knobbout, 2013), the approaches used to model AIPs are too different – also in expressive power – to compare them.

A more recent work in the normative agents area is (Criado and Such, 2017) that proposes information models and algorithms for monitoring norms under partial action observability by reconstructing unobserved actions from observed actions.

The reconstruction process entails:

1. searching for the actions that have been performed by unobserved agents; and

2. using the actions found to increase the knowledge about the state of the world.

That chapter proposes an approach that complements ours. While we assume to know in advance which events cannot be monitored, and we transform the ideal protocol into a monitorable one based on this information, the authors of (Criado and Such, 2017) "guess" the actions that the monitor could not observe, but that must have taken place because of their visible effects.

Whereas we are not aware of proposals to monitor agent interactions using commitment machines, BSPL, Splee, or HAPN under partial observability assumptions, we could mention dozens of works tackling this problem outside the MAS community.

Indeed, partial ability of monitors to observe events is a well studied problem in many contexts including command and control (Yukish et al., 1994) and runtime verification. In (Stoller et al., 2011) the authors address the problem of gaps in the observed program executions. To deal with the effects of

sampling on runtime verification, they consider event sequences as observation sequences of a Hidden Markov Model (HMM), and use an HMM model of the monitored program to fill in sampling-induced gaps in observation sequences, and extend the classic forward algorithm for HMM state estimation to compute the probability that the property is satisfied by an execution of the program[5]. Similarly to (Criado and Such, 2017), that work complements ours by estimating the likelihood of an event to occur, whereas we assume to know that likelihood, and we transform the protocol – and hence the expected sequence of observed events – based on this knowledge. Other works pursuing the objective of suitably dealing with "lossy traces" in the runtime verification area are (Basin et al., 2012; Joshi, Tchamgoue, and Fischmeister, 2017).

8.4 *The Good, the Bad and the Ugly*

The Good. Let us consider the following interaction protocol expressed in natural language:

1. Alice sends a whatsApp message to her mother Barbara asking her to buy a book (plus some implausible excuse for not doing it herself, which is not relevant for our work);

2. Barbara sends an email message to her friend Carol, responsible for the Book Shop front end, to reserve that book;

3. Carol receives Barbara email and sends a whatsApp message to Dave, the responsible of the Book Shop warehouse, to check the availability of the book and order it if necessary;

4. Dave checks if the book is available in the warehouse;

    a) if it is,

        i. he sends a whatsApp message to Emily who is in charge for physically managing the books and informing the clients if they requests can be satisfied immediately;

        ii. Emily takes the book to the front end and sends a confirmation email to Barbara telling that the book is already there;

    b) otherwise,

        i. Dave sends an email to Frank, the point of contact for the publisher of the required book, and orders it;

        ii. Frank sends a confirmation to Barbara via whatsApp telling her that the book will be available in two days.

In Section 4.2.2, we introduced the concept of event types. Through event types we can build our trace expressions on top of event sets instead of single events. This kind of generality is very useful when we want to represent

[5]Similar to what we do in Chapter 7.

generic protocols involving any kind of event. In this chapter though, we are interested in using the trace expression formalism to define AIP[6].

In particular, to make the contribution easier to understand, we consider event types representing single interaction events[7] (singleton event sets). In the following, when we write a trace expression like $ag1 \overset{msg}{\Longrightarrow} ag2{:}\tau$, we are just using a more compact syntax with respect to writing $\vartheta{:}\tau$ with $[\![\vartheta]\!] = \{ag1 \overset{msg}{\Longrightarrow} ag2\}$. In Chapter 6, we used a slightly different notation for interaction events. In the following, when we denote the interaction event $ag1 \overset{msg}{\Longrightarrow} ag2$, it is only a shortcut for ϑ, where $[\![\vartheta]\!] = \{send(ag1, ag2, msg)\}$. They are just two different ways of representing the same event; in this part of the thesis, since we are approaching more closely AIPs, we opted only for a more compact way.

Since we are interested also in the communication means used by the agents to communicate, we represent this information directly inside the interaction event, for instance, we may have $ag1 \overset{mns,cnt}{\Longrightarrow} ag2$ standing for "agent $ag1$ sends a message with content *cnt* via communication means *mns* to agent $ag2$".

$$
\begin{aligned}
P1 = {} & alice \overset{wa,buy}{\Longrightarrow} barbara{:}barbara \overset{em,reserve}{\Longrightarrow} carol{:}carol \overset{wa,checkAvail}{\Longrightarrow} dave{:} \\
& (dave \overset{wa,take2shop}{\Longrightarrow} emily{:}emily \overset{em,availNow}{\Longrightarrow} barbara{:}\epsilon \ \vee \\
& dave \overset{em,order}{\Longrightarrow} frank{:}frank \overset{wa,avail2Days}{\Longrightarrow} barbara{:}\epsilon)
\end{aligned}
$$

P1 receives the unanimous appreciation whatever the research community. In fact, two very intuitive properties are met:

1. apart from the first one, the message that each agent sends is a reaction to the message it just received, and there is an evident cause-effect link between two sequential messages;

2. in case some mutually exclusive choice must be made, the choice is up to only one agent involved in the protocol, and hence it is feasible.

These good properties take different names depending on the research communities and on the authors. The first one is named, for example, "sequentiality" (Castagna, Dezani-Ciancaglini, and Padovani, 2012), "connectedness for sequence" (Lanese et al., 2008), "explicit causality" (Singh, 2011a); the second "knowledge for choice" (Castagna, Dezani-Ciancaglini, and Padovani, 2012), "blindness" (Desai and Singh, 2008), "local choice" (Ladkin and Leue, 1995), "unique point of choice" (Lanese et al., 2008).

Meeting these two properties is closely related to the absence of covert channels; they ensure that all communications between different participants are explicitly stated, and rule out those protocols whose implementation or enactment relies on the presence of secret/invisible communications between participants: a protocol description must contain all and only the interactions

[6] We are interested in protocols where the events are only interactions.

[7] We are interested in exploiting trace expressions as AIPs, thus our events will be always agent interactions.

used to implement it (Castagna, Dezani-Ciancaglini, and Padovani, 2012). In Section 8.6, we present instead the revisited notion of Good protocol when our aim is to monitor it (and not to implement it).

The Bad. Those protocols that do not respect the connectedness for sequence and unique point of choice properties are bad, and it is not difficult to see why (see Chapter 9 for further details on how to handle these scenarios).

Let us consider protocol *P*2:

$$P2 = alice \overset{wa,buy}{\Longrightarrow} barbara{:}carol \overset{wa,checkAvail}{\Longrightarrow} dave{:} \\ (dave \overset{wa,take2shop}{\Longrightarrow} emily{:}\epsilon \ \vee \ frank \overset{wa,avail2Days}{\Longrightarrow} barbara{:}\epsilon)$$

The protocol states that *carol* can send a *checkAvail* message to *dave* only after *alice* has sent a *buy* message to *barbara*, but how can *carol* know if and when *alice* sent that message?

Also, the protocol states that either *dave* sends a message to *emily*, or *frank* sends a message to *barbara*: how can *frank* know if he is allowed to send a message to *barbara*, without coordinating with *dave* via some covert channel not shown in the protocol?

The Ugly. Protocols which are not syntactically correct are ugly, and are ignored by all the authors. However, some protocols may be ugly even if they are syntactically correct, for example if they are characterized by:

– “Causality unsafety”: consider the two shuffled sequences $carol \overset{wa,buy}{\Longrightarrow} dave{:}\epsilon \ |alice \overset{wa,buy}{\Longrightarrow} barbara{:}\epsilon$; suppose we are only able to observe what *alice* sends, and what *dave* receives. If *alice* sends *buy* and *dave* receives *buy*, we might think that the protocol above is respected. However, that observation might be due to *alice* sending *buy* to *dave*, which is not an allowed interaction: the protocol is not causality safe (Lanese et al., 2008).

– “Non-Determinism”: given an interaction taking place in some protocol state, we might want to deterministically know how to move to the next state. For example, if *alice* asks her mother to buy a book, and the protocol is

$$alice \overset{wa,buy}{\Longrightarrow} barbara : barbara \overset{wa,reserve}{\Longrightarrow} carol : \epsilon \ \vee \\ alice \overset{wa,buy}{\Longrightarrow} barbara : barbara \overset{wa,buyItYourself}{\Longrightarrow} alice : \epsilon$$

we could move on either branch. If we opt to move on the first branch, the next expected action is that *barbara* asks *carol* to reserve a book. If, instead, *barbara* tells *alice* to buy the book by herself, we have to backtrack to the previous protocol state in order to check that this interaction is allowed as well; this is extremely inefficient and should be avoided (Chapter 4).

While the notions of good and bad protocols are universally recognized, ugliness also depends on the formalism and its expressive power.

Are Bad Protocols Really so Bad? Let us suppose that the protocol is used for monitoring purposes: it does not need to be implemented or enacted. The

agents in the MAS are already there, and they are heterogeneous black boxes behaving according to their own policies and goals, in full autonomy. However, they act inside a society and they must respect the society rules, expressed as an interaction protocol. The monitor is in charge of observing messages that agents exchange, in a completely non obtrusive way, and check if they are compliant with the protocol ruling the MAS.

Let us suppose that the MAS protocol is $P1$. If the monitor is able to observe any kind of message, transmitted over any kind of communication means, we are in a standard situation, with a monitor in charge for verifying the compliance of actual interactions with a good protocol.

But what if the monitor cannot observe email messages? The protocol ruling the MAS is still $P1$, and it is still a good protocol, but from the monitor point of view it contains covert channels: the unobservable interactions taking place via email. Keeping them in the protocol would lead to false positives, as the monitor would look for messages foreseen by the protocol that it cannot see and would hence detect a protocol violation, but removing them from $P1$ leads to $P2$: a bad protocol! If the monitor observation ability is not perfect – which is an extremely realistic situation – there is no gain in struggling against bad protocols: unobservable interactions are there and generate the same problems of covert channels.

Given that good protocols where unobservable interactions have been removed can become bad protocols and that perfect observability cannot be always given for granted, we claim that – at least for monitoring purposes – bad protocols can be necessary to suitably model observable protocols.

8.5 Partial Observability: how the Good Becomes Bad

In this section we discuss how a good protocol may become (possibly) a bad one, due to unobservability or partial observability of events in the original protocol.

First of all, we need to associate with each event foreseen by the protocol, its "observability likelihood", namely the likelihood that the event can be observed by the monitor. If, when the event takes place, the monitor can always observe it, we associate 1 with the event. If the monitor can never observe the event (for example, the monitor can sniff whatsApp messages only, and the event is an email message), we associate 0 with it. If the event is transmitted over an unreliable or leaky channel, we may associate a number between 0 and 1, excluding the extremes, with it. The higher this number, the more likely the monitor will be able to observe the event when it takes place.

Let us consider $P1$ again, and let us suppose that:
(1) the observability likelihood of messages exchanged via email is 0;
(2) the observability likelihood of whatsApp messages sent by *frank* is 0.95;
(3) the observability likelihood of the other whatsApp messages is 1.

Condition 1 forces us to remove all messages exchanged via email from the protocol, and condition 3 forces us to keep all the other whatsApp messages but those sent from *frank*. The first and last conditions would lead to protocol $P2$.

The second condition, however, requires a special treatment. In fact, message $frank \stackrel{wa,okOrder}{\Longrightarrow} dave$ could be either observed or not and both cases would be correct, even if the first one should be much more frequent than the second.

The subprotocol where $frank \stackrel{wa,okOrder}{\Longrightarrow} dave$ can either take place or not can be modeled by

$$frank \stackrel{wa,okOrder}{\Longrightarrow} dave : \epsilon \ \vee \ \epsilon$$

The transformation from $P1$ to the protocol which takes observability likelihood into account requires the following steps:

1. since the observability likelihood of messages exchanged via email is 0, remove them by $P1$;
2. since the observability likelihood of the whatsApp message sent by *frank* is 0.95, substitute it with the corresponding subprotocol where the message can take place or not, and concatenate this subprotocol with the remainder;
3. since the observability likelihood of the other whatsApp messages is 1, keep them all.

The result is

$$P3 = alice \stackrel{wa,buy}{\Longrightarrow} barbara : carol \stackrel{wa,checkAvail}{\Longrightarrow} dave : \\ (dave \stackrel{wa,take2shop}{\Longrightarrow} emily : \epsilon \ \vee \ (frank \stackrel{wa,okOrder}{\Longrightarrow} dave : \epsilon \ \vee \ \epsilon) \ \cdot \ \epsilon)$$

which can be simplified into the equivalent protocol

$$P3' = alice \stackrel{wa,buy}{\Longrightarrow} barbara : carol \stackrel{wa,checkAvail}{\Longrightarrow} dave : \\ (dave \stackrel{wa,take2shop}{\Longrightarrow} emily : \epsilon \ \vee \ (frank \stackrel{wa,okOrder}{\Longrightarrow} dave : \epsilon \ \vee \ \epsilon))$$

Since dealing with likelihoods in $(0, 1)$ results into a more complex protocol, as the original protocol must be extended with the choice between observing the event or not, we might want to collapse likelihoods greater than a given threshold to 1, to avoid proliferation of choices. For this reason we assume that the protocol designer can set a threshold above which events will be considered fully observable. Let Th be such threshold and P be the protocol to transform.

P' is obtained by P applying the following rules; L is the observability likelihood of interaction *int*

1. if $L > Th$, *int* is kept;
2. if $0 < L \leq Th$, *int* is transformed into the subprotocol where *int* can either take place or not, and suitably concatenated with the remainder;
3. if $L = 0$, *int* is discarded.

Since different monitors might observe different events and observability might change over time, causing an evolution of the observable protocol, modeling the good global protocol and then transforming it based on contingencies is a better engineering approach than directly modeling the partially observable protocol. However, even if we start from a good P protocol, P' might be bad or even ugly.

8.5.1 Observability-driven transformation of trace expressions

We implemented the algorithm sketched above for protocols modeled as trace expressions. The code has been developed in SWI-Prolog and is shown below, along with comments starting with `%` that explain each clause.

The predicate that implements the algorithm is `filter_events(ProtocolToFilter, FilteredProtocol, Th, PrId)`. Since in our setting protocols are first class entities which can be analyzed, manipulated, and exchanged among agents, they are characterized by a unique name, `PrId`. `ProtocolToFilter` is the Prolog representation of the trace expression where unobservable events must be filtered out, `FilteredProtocol` is the transformation result, `Th` is the threshold above which an event is considered fully observable and hence kept in `FilteredProtocol`.

Each event type must be associated with its likelihood to be observed, thanks to the `observable(ET, Lkl, PrId)` predicate.

As an example, if we had a parametric English Auction interaction protocol where the `buy(X)` parametric event type observability is `0.5`, and the observability of all the other event types is `1`, we would write

```
observable(buy(X), 0.5, english_auct) :- !.
observable(E, 1, english_auct) :- E \= buy(_).
```

where `!` prevents the Prolog interpreter from backtracking when it is executed[8] and `\=` stands for "cannot unify with". Uppercase symbols represent logical variables, and `p :- q, r, s.` should be read as "if q, r, s hold, then p holds".

`filter_events` operates according to the trace expression syntax. We omit the rule for dealing with parameters.

```
filter_events(epsilon, epsilon, _, _) :- !.
% empty trace expression epsilon is transformed into epsilon

filter_events(ET:T, TFiltered, Th, PrId) :-
  observable(ET, 0, PrId), !,
  filter_events(T, TFiltered, Th, PrId).
% trace expression ET:T where observable(ET, 0, PrId) is
% transformed into TFiltered if T is transformed into
% TFiltered (ET is removed)

filter_events(ET:T, TFiltered, Th, PrId) :-
```

[8] The functioning of "cut" is more complex than this, but we can ignore the details.

```
  observable(ET, Lkl, PrId), Lkl > Th, !,
  filter_events(T, T1, Th, PrId),
  TFiltered = ET:T1, !.
% trace expression ET:T where observable(ET, Lkl, PrId) and
% Lkl > Th is transformed into ET:T1, where T1 results from
% transforming T (ET is kept)

filter_events(ET:T, TFiltered, Th, PrId) :-
  observable(ET, Lkl, PrId),
  filter_events(T, T1, Th, PrId), Lkl =< Th,
  TFiltered = ((ET:epsilon) \/ epsilon) * T1), !.
% in trace expression ET:T where observable(ET, Lkl, PrId) and
% Lkl <= Th, ET becomes optional.
% ET:T becomes ((ET:epsilon)\/epsilon)*T

filter_events(T1\/T2, TFiltered, Th, PrId) :-
  filter_events(T1, TFiltered1, Th, PrId),
  filter_events(T2, TFiltered2, Th, PrId),
  TFiltered = (TFiltered1 \/ TFiltered2), !.
% filtering T1\/T2 means filtering T1, filtering T2, and
% joining the results with the \/ operator.
% The same holds for the other operators, |, *, /\ (not shown)
```

The code for `filter_events` is 36 lines long and – provided a basic knowledge of logic programming – is self-explaining. Despite its simplicity, it can operate on very complex parametric and recursive (also non terminating) protocols. The magic behind the "invisible" management of non terminating protocols like *P4* is the use of the SWI-Prolog coinduction library which allows to cope with infinite terms without entering into loops[9].

8.5.2 Implementation and Experiments

We have experimented the filtering algorithm on the parametric trace expression modeling the English Auction protocol presented in Section 6.4. As anticipated, we initially assumed that the only partially observable event was `buy(X)` with observability likelihood `0.5`. By setting the threshold to `0.7`, all occurrences of `buy(X)` became optional, while with a threshold equal to `0.4` they were all kept in the protocol. By setting the observability likelihood of `buy(X)` to `0`, any occurrence of `buy(X)` was removed from the protocol.

The algorithm was also run on a variant of the Alternating Bit Protocol (Deniélou and Yoshida, 2012) with 6 agents.

Different observability likelihoods and different thresholds were set with both protocols, to test the algorithm in an exhaustive way.

[9]SWI-Prolog implementation available open-source at:
`https://github.com/AngeloFerrando/TExpSWIPrologConnector/tree/master/TExpSWIPrologConnector/src/main/resources/prolog-code`

8.6 Revisiting Good and Bad notions

Now that we have introduced the concept of good, bad and ugly protocols, we have to focus on their use and how this can influence our notions. When we introduced these contents we compared them with the corresponding properties in the other research communities. The reason for that concerns the possible uses of the protocols. Once we define an AIP using a trace expression, we can use such definition to guide the development of a MAS (enact the protocol) or to guide the monitoring of a MAS (monitor the protocol).

The concepts of Good, Bad and Ugly AIPs in the sections above are very strong. When we have a sequence of messages we require a cause-effect link between the messages, and when we have choices, we always require they are taken locally by one agent. These requirements are so strong because we wanted to present a good level of classification that would be suitable both for enactment and monitoring purposes. In particular, when we talk about implementation and monitoring of the protocols, we should also well establish what kind of communication model will be used. The choice of a specific communication model influences what is a Good and a Bad protocol. Until now, we have not assumed any particular communication model for the implementation of all communicative acts. Consequently, we have been considering the least restrictive one; the asynchronous model, where the messages are decomposed in their sending and receiving parts, and they can be reordered and lost. Because of this, the requirements to be Good are very strong.

When in Chapters 9 and 10 we will use AIPs for monitoring purposes, we will need a more relax notion of Good and Bad protocols, adding some constraints on the communication model used. More specifically, we are going to consider a communication model where each send event is immediately followed by its corresponding receive event. This kind of model can be called *Realizable with Synchronous Communication* (RSC) (Chevrou, Hurault, and Quéinnec, 2016). If the couple (send event, corresponding receive event) is viewed atomically, this corresponds to a synchronous communication execution. If the MAS exploits RSC for achieving the communication among the agents, we can relax the constraints for an AIP to be considered Good. We can reformulate the properties that makes a protocol Good as follows:

1. two sequential messages must always have at least one agent in common;
2. in case some mutually exclusive choice must be made, the two messages involved in this choice must have at least one agent in common.

Since we know that the sending of a message is always followed by its corresponding receipt, we can relax both the constraints, because we can consider the messages as atomic interactions, where observing the send or receive events is the same.

Considering as an example the following trace expression

$$\tau = alice \stackrel{msg1}{\Longrightarrow} bob : charlie \stackrel{msg2}{\Longrightarrow} bob{:}\epsilon$$

if we follow the definition of Good and Bad protocols (Section 8.4), we conclude that this is a Bad protocol, because *msg*2 is not a reaction to *msg*1. There is not an evident cause-effect link between the two messages. But, knowing that we are using a RSC communication model, we can use the reformulated constraints concluding that is a Good one. In fact, even though there is no cause-effect between *msg*1 and *msg*2, since they are two atomic interactions, it is enough to look at *bob* in order to make sure that the order of the messages is preserved.

In the same way, if we consider this other example

$$\tau = (alice \overset{msg1}{\Longrightarrow} bob{:}\epsilon) \vee (charlie \overset{msg2}{\Longrightarrow} bob{:}\epsilon)$$

we have a similar situation. With respect to the definition given in Section 8.4, also this protocol is a Bad one, because the choice is not up to only one agent involved in the protocol (both *alice* and *charlie* can choose). But, since we know that the messages are atomic interactions, again it is enough to look at *bob* in order to make sure that the order of the messages is preserved.

8.7 Discussion

In this chapter we have analyzed the notions of good, bad and ugly protocols inside and outside the MAS community, and we have motivated the reason why bad protocols are not that bad by considering runtime monitoring of MAS with unobservable events.

The likelihood of the events has been also integrated inside the RIVERtools framework (Chapter 14), and so the `filter_events` predicate. Consequently, the developer can write freely the AIPs and the framework will automatically identify each of them as a Good, Bad or Ugly protocols.

In the following chapters, we consider two interesting aspects that derives from the issues raised by this chapter. In Chapter 9, we show how to handle bad protocols when we are interested in decentralized the runtime verification process. In Chapter 11, we study and propose a conformance algorithm for AIPs represented through trace expressions. In both scenarios, all the trace expressions used for the examples have been already filtered with respect to the likelihood of the events. Consequently, these trace expressions are already exploited in their specific context. In Chapter 9, in order to see how to handle a bad protocol (even though we started from a good one) in a decentralize scenario, and in Chapter 11, in order to show how the exploited trace expressions can be compared for reusing agents.

9 *Decentralized Runtime Verification of Agent Interaction Protocols*

"The future is already here – it's just not evenly distributed."
- William Gibson

In the previous chapter we presented how to use trace expressions to define AIP. Until now in the thesis we have always focused our intentions on the centralized runtime verification of the specifications. In this chapter we present how to obtain the same result in a decentralized way. In particular, we describe DecAMon, an algorithm for decentralizing the monitoring of the MAS communicative behavior. If some agents in the MAS are grouped together and monitored by the same monitor, instead of individually, a partial decentralization of the monitoring activity can still be obtained even if the "unique point of choice" (a.k.a. local choice) and "connectedness for sequence" (a.k.a. causality) coherence conditions are not satisfied by the protocol (Chapter 8). Given an AIP specification, DecAMon outputs a set of "Monitoring Safe Partitions" of the agents, namely partitions P which ensure that having one monitor in charge for each group of agents in P allows detection of all and only the protocol violations that a fully centralized monitor would detect.

The contents of this chapter are published in (Ferrando, Ancona, and Mascardi, 2017)

9.1 Introduction

When the system is small, one centralized monitor can check it all without becoming a bottleneck. Nevertheless, centralized monitoring does not scale with the growth of the system dimension and a decentralized monitoring approach may be the only viable solution for coping with the system complexity. Also, decentralized monitoring may be a more natural choice when the system is distributed, since different monitors can be associated with groups of physically, geographically, or logically connected entities, gaining in efficiency and modularity.

Associating an individual agent or group of agents with a sniffer in charge of observing their communicative behavior does not raise serious technical problems if JADE or Jason are used (Chapter 14). Rather, the actual problem for dynamically verifying that a MAS behaves according to a given AIP is

how to ensure that the system made up of the decentralized monitors detects all and only the same protocol violations that a single centralized monitor observing the MAS would detect?

In this chapter we describe *DecAMon*, an algorithm for *Dec*entralizing the *A*gent system *Mon*itoring which works also in case the global AIP specification, expressed using trace expressions (Chapter 4), does not satisfy those coherence conditions.

This proposal, which falls in the Decentralized Runtime Verification area (Bonakdarpour et al., 2016b), is based on the idea that, between a fully centralized and a fully decentralized monitoring approach, a third viable solution is possible: partially decentralized monitoring.

DecAMon is completely agnostic w.r.t. the type of observed events. For sake of presentation clarity, in this chapter, we limit ourselves to consider interaction events, the *DecAMon* algorithm can be applied to trace expressions dealing with events of any kind. The only requirement for the *DecAMon* algorithm to work is that the set of agents involved in a given event should be efficiently computed.

9.2 Motivations

Alice and Carol are two PhD students. They are writing a paper for the AAMAS conference with their colleague Bob and their supervisor Dave. Alice and Carol are in charge for running experiments. They are working on a shared repository and they agree on the notion of satisfactory results. A few weeks before the deadline, they decide that if the experimental results will reach a satisfactory level by the evening, Alice will contact their supervisor to meet, otherwise Carol will ask Bob to meet, to fix the problems.

If Alice and Carol are modeled as software agents, the resulting global agent interaction protocol, AIP_1, can be represented by $alice \overset{meet}{\Longrightarrow} dave{:}\epsilon \vee carol \overset{meet}{\Longrightarrow}$

$bob{:}\epsilon$ where, as we presented in Chapter 8, $ag1 \stackrel{msg}{\Longrightarrow} ag2$ stands for "there is an interaction between $ag1$ and $ag2$, with exchanged message msg"[1].

Once the paper is ready, Alice and Carol decide that Alice will make a first submission in the morning and then she will make a last check and, eventually, modify and re-submit the paper. Carol will send the final version to their supervisor in the evening. The resulting protocol can be modeled by the trace expression

$$\text{AIP}_2 = alice \stackrel{submit}{\Longrightarrow} aamas : (alice \stackrel{submit}{\Longrightarrow} aamas : \epsilon \ \vee \ \epsilon) \ \cdot \ carol \stackrel{submitted}{\Longrightarrow} dave : \epsilon$$

where Alice makes one or two submissions and after Carol sends *submitted* to Dave.

Even if AIP 1 and AIP 2 are simple and realistic, they have two major problems.

The first problem is that these protocols are too abstract. Although their meaning can be easily grasped, what does actually mean for two agents to interact? Let us consider this simple example: $alice \stackrel{m1}{\Longrightarrow} bob : alice \stackrel{m2}{\Longrightarrow} carol : \epsilon$. Once Alice has sent $m1$ to Bob, could she immediately send $m2$ to Carol, without worrying if Bob received $m1$, or not? This problem is related with the granularity of interactions, which describe message sending and receiving as an atomic action. Since, in many cases, communication is asynchronous, assuming interaction atomicity is not always the case. We should decouple the sending event and the reception event, in order to provide a more precise description of the system. By using $ag1_{\langle \stackrel{msg}{\Longrightarrow} ag2 \rangle}$ to denote "$ag1$ sends msg to $ag2$" and $_{\langle ag1 \stackrel{msg}{\Longrightarrow} \rangle} ag2$ to denote "$ag2$ receives msg from $ag1$", we can describe both a synchronous behavior, where nothing should happen between a sending and the corresponding reception (we use a for *alice*, b for *bob*, and c for *carol*), $\text{CAIP}_1 = a_{\langle \stackrel{m1}{\Longrightarrow} b \rangle} : {}_{\langle a \stackrel{m1}{\Longrightarrow} \rangle} b : a_{\langle \stackrel{m2}{\Longrightarrow} c \rangle} : {}_{\langle a \stackrel{m2}{\Longrightarrow} \rangle} c : \epsilon$, and an asynchronous one, where sending comes first, the corresponding reception comes after, but other events could take place in between, $\text{CAIP}_2 = a_{\langle \stackrel{m1}{\Longrightarrow} b \rangle} : ({}_{\langle a \stackrel{m1}{\Longrightarrow} \rangle} b : \epsilon \ | \ a_{\langle \stackrel{m2}{\Longrightarrow} c \rangle} : {}_{\langle a \stackrel{m2}{\Longrightarrow} \rangle} c : \epsilon)$[2]. Of course, ordering among events in each branch must be preserved. The idea behind $\tau_1 | \tau_2$ is that τ_1 and τ_2 are independent.

The second problem, is that both AIP 1 and AIP 2 are bad protocols according to the definition provided in 8. Moving from a global protocol that involves all the agents in the MAS to a view of that protocol restricted to an agent subset is usually named "projection". In Section 8.2 we already presented the concept of projection of a trace expression.

Even if we are able to project onto individual agents, how can we be sure that the individual monitoring gives the same results as the centralized one? This second problem is independent from the granularity of communicative events. Rather, it depends on taking a centralized or a decentralized view point. Let us consider AIP_1. A monitor M_{alice} associated with Alice and driven by the

[1]Instead of having a generic event we focus on an interaction one like we did in Chapter 8.
[2]We name these protocols $CAIP_1$ and $CAIP_2$ to stress that they are "*C*oncrete".

protocol portion that involves Alice only, will consider her behavior correct if she sends a *meet* message to Dave. In the same way, a monitor M_{carol} will consider Carol's behavior correct if she sends a *meet* message to Bob. However, performing both actions will not be compliant with AIP_1 as they are mutually exclusive. Unfortunately, none among M_{alice}, M_{carol}, M_{bob}, M_{dave} alone can verify if mutual exclusivity is respected. Following the terminology introduced in Section 8.4, AIP 1 does not satisfy the unique point of choice coherence condition.

The problem with AIP_2 is different: what happens if Carol sends *submitted* to Dave before Alice submits the paper? The AIP_2 portion that M_{carol} sees is

$$carol \overset{submitted}{\Longrightarrow} dave : \epsilon$$

On the other hand, the portion seen by M_{alice} is

$$alice \overset{submit}{\Longrightarrow} aamas : (alice \overset{submit}{\Longrightarrow} aamas : \epsilon \ \vee \ \epsilon)$$

No individual monitor can state whether a *submitted* message sent by Carol to Dave comes after Alice completed her last submission. According to the behavioral types terminology (Section 8.4), AIP_2 does not satisfy the *connectedness for sequence* coherence condition. According to Desai and Singh's terminology (Chapter 8), AIP_2 problem is due to the *blindness* of *carol* w.r.t. the *submit* message that *alice* sends to *aamas*.

Such blindness can be related to the presence of private channels that cannot be observed by the monitors, or can be related to partial observability issues. In Section 8.5, we showed indeed how a good protocol can become a bad one because of partial observability of communication channels. Whatever the reasons, monitoring bad protocols cannot always be decentralized[3] on any possible subset of the agents. Instead, a protocol that meets the coherence conditions can always be fully decentralized since protocol violations are only due to messages which are exchanged in a given protocol state, but were not allowed in that state. For example $AIP_4 = alice \overset{submit}{\Longrightarrow} aamas : (alice \overset{sumbitted}{\Longrightarrow} dave : \epsilon \mid aamas \overset{ack}{\Longrightarrow} alice : \epsilon)$ could be violated by Alice submitting the paper twice. However, the second $alice \overset{submit}{\Longrightarrow} aamas$ interaction would violate both $\Pi(AIP_4, \{alice\})$ and $\Pi(AIP_4, \{aamas\})$ (the projections[4] of the global protocol AIP_4 onto $\{alice\}$ and $\{aamas\}$, respectively), so at least one decentralized monitor between M_{alice} and M_{aamas} would immediately detect it. A protocol that does not meet the coherence conditions causes problems only when we try to fully decentralize its monitoring: each agent *ag* has its own monitor that checks if *ag* behavior is compliant with $\Pi(\tau, \{ag\})$ (Section 4.4). This may cause the loss of sequentiality and mutual exclusivity constraints. As long as we assume that centralized monitoring takes place no problems arise, apart from the enormous bottleneck that the centralized monitor may become!

Given a protocol specification and the set *Ags* of agents in the MAS, *DecAMon* faces the partial decentralization problem by computing a set of "Monitoring Safe (MS) partitions" of *Ags*. If a violation of the behavior patterns defined

[3] We consider the revisited notion of bad protocol that will be presented in Section 8.6.
[4] Section 8.2.

by the protocol takes place, one monitor in charge for one group in the MS partition will detect it.

9.3 DecAMon: a Gentle Introduction

Let us suppose that the agents involved in the MAS are *alice*, *bob*, *carol*, and *dave*.

If $\{\{alice, carol\}, \{bob\}, \{dave\}\}$ is a MS partition, then *alice* and *carol* must be monitored by the same monitor $M_{\{alice, carol\}}$, whereas *bob* and *dave* may be monitored by distinct monitors. This does not mean that having one monitor $M_{\{alice, carol\}}$ for *alice* and *carol* and one $M_{\{bob, dave\}}$ for *bob* and *dave* (to be monitored together), or one single monitor $M_{\{alice, bob, carol, dave\}}$ for all the four agents, is not monitoring safe: larger groups can be formed, provided that those agents which must stay together, are monitored together. The above partition is one of those returned by *DecAMon* on the AIP_1 protocol introduced in Section 9.2: if the same monitor observes both *alice* and *carol*, it will be able to detect violations of mutual exclusivity between $alice \stackrel{meet}{\Longrightarrow} dave$ and $carol \stackrel{meet}{\Longrightarrow} bob$.

In a similar way, one MS partition of the agents involved in AIP_2 is $\{\{alice, dave\}, \{aamas\}, \{carol\}\}$: if the same monitor is in charge for both *alice* and *dave*, it can verify that the interaction involving *dave* (and *carol*) takes place after the interactions involving *alice* (and *aamas*).

Intuition 1 (Monitoring Safety (MS)). *A partition of Ags P is Monitoring Safe (MS partition, abbreviated in MS in the sequel) if it enjoys the following property: if the agents belonging to the same group in P are monitored together, no loss of sequentiality and mutual exclusivity constraints takes place; one among the decentralized monitors detects a violation of "its portion" of the global protocol iff a violation of the global protocol occurs.*

If the system monitoring cannot be decentralized, *DecAMon* will return only one MS, $\{Ags\}$. On the other hand, if each agent $ag_i \in Ags$, with $i \in \{1, \ldots, n\}$ can be monitored independently from the others, *DecAMon* will output $\{\{ag_1\}, \{ag_2\}, \{ag_3\}, \ldots, \{ag_n\}\}$.

DecAMon agnosticism w.r.t. the events syntax gives us the flexibility to execute it on abstract and concrete agent protocol specifications by defining the *involved* function as

$$involved(ag1 \stackrel{msg}{\Longrightarrow} ag2) = \{ag1, ag2\}$$
$$involved(ag1_{\langle \stackrel{msg}{\Longrightarrow} ag2 \rangle}) = \{ag1\}$$
$$involved(_{\langle ag1 \stackrel{msg}{\Longrightarrow} \rangle} ag2) = \{ag2\}.$$

When we adopt a concrete protocol perspective, where sending and reception are distinct, the only entity involved in a message sending (resp. reception) is the sender (resp. the receiver), even if we keep track of the message sender also in a "receive" event $_{\langle ag1 \stackrel{msg}{\Longrightarrow} \rangle} ag2$ and viceversa.

Continuing the AIP_1 example, the other MSs are

$$\{\{alice\}, \{carol\}, \{bob, dave\}\}$$

$\{\{alice, bob\}, \{carol\}, \{dave\}\}$
$\{\{alice\}, \{carol, dave\}, \{bob\}\}$

If e_1 and e_2 are joined by an $\vee$ operator in the AIP like $alice \overset{meet}{\Longrightarrow} dave$ and $carol \overset{meet}{\Longrightarrow} bob$ in AIP_1, and $involved(ev_1)$ has empty intersection with $involved(e_2)$, one agent $ag1 \in involved(e_1)$ must be monitored together with one agent $ag2 \in involved(e_2)$ to ensure that mutual exclusivity becomes verifiable. In a similar way, if e_1 and e_2 are two sequential events like $alice \overset{submit}{\Longrightarrow} aamas$ and $carol \overset{submitted}{\Longrightarrow} dave$ in AIP_2, and $involved(e_1)$ has empty intersection with $involved(e_2)$, one agent $ag1 \in involved(e_1)$ must be monitored together with one agent $ag2 \in involved(e_2)$ in order to verify the correct sequentiality of e_1 and e_2. In both cases, if there exists one agent $agI \in involved(e_1) \cap involved(e_2)$ no grouping is required: the monitor associated with agI can verify mutual exclusiveness and correct sequencing between e_1 and e_2.

9.3.1 High-level Description and Examples

Now, we are ready to introduce the notions of critical point of a trace expression and of minimality of a MS partition. The function $first(\tau)$ returns all the first events of τ and $last(\tau)$ all its last events. For example, $first(a : \epsilon \vee b : c : \epsilon) = \{a, b\}$ and $last(a : \epsilon \vee b : c : \epsilon) = \{a, c\}$. Their precise definition is given in Section 9.4 and 9.5.

Definition 7 (Critical Point)**.** *A couple of events* (e_1, e_2) *is a critical point of* τ *iff* τ_{sub} *is a sub-expression of* τ *such that*

- $\tau_{sub} = \tau_1 \vee \tau_2$ *and* $e_1 \in first(\tau_1)$, $e_2 \in first(\tau_2)$ *and* $involved(e_1) \cap involved(e_2) = \emptyset$, *or*
- $\tau_{sub} = e_1{:}\tau_2$ *and* $e_2 \in first(\tau_2)$ *and* $involved(e_1) \cap involved(e_2) = \emptyset$, *or*
- $\tau_{sub} = \tau_1 \cdot \tau_2$ *and* $e_1 \in last(\tau_1)$, $e_2 \in first(\tau_2)$ *and* $involved(e_1) \cap involved(e_2) = \emptyset$.

We say that τ_{sub} *generates the critical point* (e_1, e_2). *Uniqueness of* (e_1, e_2) *is violated if both* e_1 *and* e_2 *take place. Sequentiality of* (e_1, e_2) *is violated if* e_2 *takes place before* e_1.

Definition 8 (Minimal Monitoring Safety (MMS))**.** *The partition P of agents Ags is Minimal Monitoring Safe (MMS) if it is Monitoring Safe and if splitting one of the groups of agents in P leads to a partition that does not satisfies monitoring safety any longer.*

For generating a set of MSs, *DecAMon* exploits *merge*:

$$merge : \mathcal{P}(\mathcal{P}(Ags)) \times \mathcal{P}(\mathcal{P}(Ags)) \rightarrow \mathcal{P}(\mathcal{P}(Ags))$$

One argument C of *merge* (no matter which, since merge is commutative) consists of new groups of agents that are constrained to be monitored together; the other argument *OldC* models the existing agent grouping constraints: we name them “constraint stores”.

The result of *merge* is a new constraint store $NewC$ where both the constraints in $OldC$ and those in C hold. No unnecessary constraints (namely, no unnecessary groupings) are added to the *merge* result. The way *merge* works ensures that the groups of agents in $NewC \in \mathcal{P}(\mathcal{P}(Ags))$ will be disjoint if the groups of agents in C were disjoints, and the groups of agents in $OldC$ were. Let us introduce *merge* by means of an example: the idea behind

$$merge(\{\{ag1, ag2\}\}, \{\{ag1, ag3\}, \{ag4, ag5\}\})$$

is to add the new constraint "agents $ag1$ and $ag2$ must be monitored together" to the constraint store $\{\{ag1, ag3\}, \{ag4, ag5\}\}$ stating that $ag1$ and $ag3$ must be monitored together, as well as $ag4$ and $ag5$. The only constraint store resulting from this merge is $NewC = \{\{ag1, ag2, ag3\}, \{ag4, ag5\}\}$, where both the previous and the new constraints are respected. The amount of agents that are constrained to be monitored together is minimized: in $NewC$, $ag1$ and $ag4$ can be monitored independently as there is no reason to group them. The constraint store $NewC' = \{\{ag1, ag2, ag3, ag4, ag5\}\}$ satisfies the old and new constraints as well but uselessly imposes that $ag1$ and $ag4$ are monitored together: *merge* will never return it.

DecAMon carries out a structural analysis of the trace expression in order to find those agents that must be monitored together because they are involved in a critical point. As soon as new groups of agents that must be monitored together are found, the new constraint store is merged with the previously computed one: given a trace expression $\tau = \tau_1\ op\ \tau_2$ where op is a binary operator, *DecAMon* computes the constraint stores due to op (they may be more than one, as shown in the sequel) and computes the combinations obtained by merging each of them with each of those resulting from τ_1 and each of those resulting from τ_2. Since the constraint stores deriving from τ_1 and τ_2 are computed independently, they could overlap up to some extent and their merge could generate groupings with unnecessary constraints. To cope with this problem we have implemented a post-processing algorithm that allows us to obtain the set of MMSs as a refinement of the *DecAMon* output, by removing those MSs that add useless constraints to other MSs. The global minimality property can be obtained either via this post-processing activity where each returned MS is compared with all the others, or by making *merge* more complex (*merge* would need to know all the possible MSs for each trace expression branch to discard the overlapping ones and return only minimal MSs). We opted for the first solution.

Let us consider another example: if we had to compute

$$merge(\{\{ag1, ag2, ag4\}\}, \{\{ag1, ag3\}, \{ag2, ag5\}\})$$

the only possible result would be to merge $\{ag1, ag3\}$ and $\{ag2, ag5\}$ where $ag1$ and $ag2$ could be monitored independently, to meet the new constraint where they must be monitored together; $a4$ must be grouped with $a1$ and $a2$. The result is $\{\{ag1, ag2, ag3, ag4, ag5\}\}$.

Let us consider the more complex protocol AIP_5 defined as

$$(ab{:}bc{:}\epsilon \mid de{:}ef{:}\epsilon \mid gh{:}hi{:}\epsilon) \cdot (jk{:}\epsilon \mid lm{:}\epsilon \mid no{:}\epsilon)$$

where ab stands for $a \stackrel{ab}{\Longrightarrow} b$, bc stands for $b \stackrel{bc}{\Longrightarrow} c$, and so on.

DecAMon starts exploring AIP$_5$ looking for critical points. The outmost AIP$_5$ operator is a concatenation, which may generate critical points. *DecAMon* computes all the last events in $\tau_1 = (ab{:}bc{:}\epsilon \mid de{:}ef{:}\epsilon \mid gh{:}hi{:}\epsilon)$ and all the first events in $\tau_2 = (jk{:}\epsilon \mid lm{:}\epsilon \mid no{:}\epsilon)$. They turn out to be $last(\tau_1) = \{bc, ef, hi\}$ and $first(\tau_2) = \{jk, lm, no\}$.

Any couple (e_1, ev_2) s.t. $e_1 \in last(\tau_1)$, $e_2 \in first(\tau_2)$, and $involved(e_1) \cap involved(e_2) = \emptyset$ is a critical point. Here, (bc, jk), (bc, lm), (bc, no), (ef, jk), (ef, lm), (ef, no), (hi, jk), (hi, lm), (hi, no), are all the critical points generated by the outmost $\cdot$ in AIP$_5$.

For each critical point (e_1, e_2), one agent involved in e_1 must be grouped together with one agent involved in e_2.

Definition 9 (Critical Point Satisfaction)**.** *A group of agents satisfies a critical point (e_1, e_2) if it contains one agent involved in e_1 and one agent involved in e_2.*

Definition 10 (Trace Expression Satisfaction)**.** *A constraint store C satisfies a trace expression if all the critical points generated by the outmost operator in that trace expression are satisfied by one group in C.*

To make another example, $C5_1 = \{\{b, e, h, j, l, n\}\}$ satisfies AIP$_5 = \tau_1 \cdot \tau_2$ since b which is involved in bc is grouped with j involved in jk, l involved in lm, and n involved in no. The same holds for e involved in ef and h involved in hi.

Also $C5_2 = \{\{b, k, m, o\}, \{e, h, j, l, n\}\}$, satisfies AIP$_5$: b is grouped with k, m, and o hence satisfying (bc, jk), (bc, lm), and (bc, no); e and h are grouped with j, l, n, hence satisfying (ef, jk), (ef, lm), (ef, no), (hi, jk), (hi, lm), and (hi, no).

The same holds for $C5_3 = \{\{c, k\}, \{b, m, o\}, \{e, h, j, l, n\}\}$: $\{c, k\}$ satisfies (bc, jk); $\{b, m, o\}$ satisfies (bc, lm) and (bc, no); $\{e, h, j, l, n\}$ satisfies all the remaining critical points.

The constraint store

$$\{\{c, j\}, \{f, l\}, \{i, n\}, \{a\}, \{b\}, \{d\}, \{e\}, \{g\}, \{h\}, \{k\}, \{m\}, \{o\}\}$$

instead, does not satisfy AIP$_5$: for example, no group satisfies (bc, lm).

We define $C\tau_0$ as the constraint store that contains all and only one singleton set $\{ag\}$ for each agent ag involved in τ. Given the initial constraint store $C5_0$ for the protocol AIP$_5$, *DecAMon* merges $C5_0$ with one of the constraint stores $C5_i$ that satisfy AIP$_5$, selected on a nondeterministic basis. Then, it recursively explores the components τ_1 and τ_2 of AIP$_5$ and adds the newly discovered constraints to the previously computed constraint store. The sequences $ab{:}bc{:}\epsilon$, $de{:}ef\epsilon$ and $gh{:}hi{:}\epsilon$ in τ_1 do not generate any new critical point because they verify the connectedness for sequence condition. Moreover, they are joined by a shuffle operator that generates no critical points. Thus, no new constraints are generated because of τ_1. In a similar way, no new constraints are generated because of τ_2.

The nondeterministic selection of one of the constraint stores satisfying the currently analyzed trace expression is repeated for each possible constraint

stores. By backtracking to any point of choice, *DecAMon* can produce all the possible MSs, one at a time: $C5_1$, $C5_2$, $C5_3$, $C5_4$, together with other 5628 possible initial constraint stores, are the MSs output by *DecAMon*!

If τ_1 were $(ab{:}cd{:}\epsilon \mid ef{:}gh{:}\epsilon)$ (AIP_6) the new constraints $\{\{a,c\},\{e,g\}\}$ (or $\{\{a,d\},\{f,g\}\}$, ...) due to connectedness for sequence violation should have been merged with $C5_i$ giving a different (and smaller) final set of MSs.

If τ_1 were $(ab{:}cd{:}\epsilon \vee ef{:}gh{:}\epsilon)$ (AIP_7) a further constraint $\{a,e\}$ (or $\{a,f\}$, or $\{b,e\}$, or $\{b,f\}$) due to unique point of choice violation should have been merged with the previous ones.

Finally, let us consider the concrete protocol $CAIP_1 = a_{\langle \stackrel{m1}{\Longrightarrow} b\rangle} : {}_{\langle a \stackrel{m1}{\Longrightarrow}\rangle} b : a_{\langle \stackrel{m2}{\Longrightarrow} c\rangle} : {}_{\langle a \stackrel{m2}{\Longrightarrow}\rangle} c : \epsilon$. As anticipated, *DecAMon* works exactly in the same way, provided it can compute the *involved* function. $CAIP_1$ can be seen as $a_{\langle \stackrel{m1}{\Longrightarrow} b\rangle} : \tau_1$. Its outmost operator is the first prefix with $\{a_{\langle \stackrel{m1}{\Longrightarrow} b\rangle}\}$ at its left, $first(\tau_1) = \{{}_{\langle a \stackrel{m1}{\Longrightarrow}\rangle} b\}$ and these two events share no involved agents: $(a_{\langle \stackrel{m1}{\Longrightarrow} b\rangle}, {}_{\langle a \stackrel{m1}{\Longrightarrow}\rangle} b)$ is a critical point. The constraint store generated by $CAIP_1$ is $\{\{a,b\}\}$ which must be merged with the initial constraint store $\{\{a\},\{b\},\{c\}\}$ leading to $\{\{a,b\},\{c\}\}$. Now *DecAMon* is called on $\tau_1 = {}_{\langle a \stackrel{m1}{\Longrightarrow}\rangle} b : a_{\langle \stackrel{m2}{\Longrightarrow} c\rangle} : {}_{\langle a \stackrel{m2}{\Longrightarrow}\rangle} c : \epsilon$ generating the new constraint store $\{\{a,c\}\}$ which must be merged with $\{\{a,b\},\{c\}\}$ leading to $\{\{a,b,c\}\}$. In the end, all the three agents must be monitored together. This is correct: how can M_b alone verify that when b receives $m1$ from a, a actually sent it before? If we make either security assumptions ("all the received messages have been actually sent by the sender") or strong assumptions on the underlying network reliability ("all the sent messages will be received", or even "all the sent messages will be received in the same order they were sent") we can relax some monitoring safety constraints, but this is not possible in general.

9.4 Design

The definition of *first* does not need to take cycles – which are due to trace expressions recursive definitions – into account; in fact, contractiveness ensures that, while exploring a trace expression following its syntactical structure, a prefix operator will be met in a finite number of steps.

- $first(\epsilon) = \{\}$
- $first(\vartheta{:}\tau) = \{\vartheta\}$
- $first(\tau_1 \cdot \tau_2) = first(\tau_1) \cup first(\tau_2)$ if $\varepsilon(\tau_1)$;
  $first(\tau_1 \cdot \tau_2) = first(\tau_1)$ otherwise
- $first(\tau_1 \wedge \tau_2) = first(\tau_1 \vee \tau_2) = first(\tau_1 | \tau_2) =$
  $first(\tau_1) \cup first(\tau_2)$.

The definition of *last* is more complex because it must not recall itself in case of cyclic trace expressions and contractiveness is not enough to avoid entering a loop. For example, $\tau = e{:}\tau$ is contractive, but we must have plenty of time and patience if we are going to look for its last element! In this case,

last should return {} (and we should do the same...) but it can do this only if it keeps track of the already met trace expressions. To this aim *last* saves the argument of each call into a global repository; if it is called on τ and it had already been called on τ before, it returns {}:

- $last(\epsilon) = \{\}$
- $last(\tau) = \{\}$ if *last* had already been called on τ;
  otherwise, the following rules apply:
  - $last(\vartheta{:}\tau) = last(\tau) \cup \{\vartheta\}$ if $\varepsilon(\tau)$;
    $last(\vartheta{:}\tau) = last(\tau)$ otherwise
  - $last(\tau_1 \cdot \tau_2) = last(\tau_2)$
  - $last(\tau_1 \wedge \tau_2) = last(\tau_1 \vee \tau_2) = last(\tau_1 | \tau_2) =$
    $last(\tau_1) \cup last(\tau_2)$.

To describe *DecAMon* we first describe the *DecOne* logical predicate

$$DecOne \subseteq \mathcal{T} \times \mathcal{P}(\mathcal{P}(Ags)) \times \mathcal{P}(\mathcal{P}(Ags))$$

The way *DecOne* works ensures that the groups of agents in $Arg \in \mathcal{P}(\mathcal{P}(Ags))$ are disjoint, for each *Arg* that can appear as its second or third argument. Given a trace expression $\tau \in \mathcal{T}$, $DecOne(\tau, OldC, NewC)$ holds iff there exists a constraint store C s.t. C satisfies τ and $NewC = merge(OldC, C)$. $DecOne(\tau, OldC, NewC)$ nondeterministically selects one of the constraint stores that satisfy τ, let us name it C, and merges it with *OldC* resulting into *NewC*. Since *DecOne* must avoid entering loops, it operates like *last* keeping track of the already met trace expressions.

• $DecOne(\epsilon, OldC, OldC)$

• $DecOne(\tau, OldC, OldC)$ if *DecOne* had already been called on τ; otherwise, the following rules apply:

i. $DecOne(\vartheta{:}\tau, OldC, NewC)$ iff
   $\exists\, C'$ s.t. $DecOne(\tau, OldC, C')$,
   $\exists\, C$ that satisfies $\vartheta{:}\tau$, and
   $NewC = merge(C', C)$;

ii. $DecOne(\tau_1 \cdot \tau_2, OldC, NewC)$
   (resp. $DecOne(\tau_1 \vee \tau_2, OldC, NewC)$) iff
   $\exists\, C_1$ s.t. $DecOne(\tau_1, OldC, C_1)$,
   $\exists\, C_2$ s.t. $DecOne(\tau_2, C_1, C_2)$,
   $\exists\, C$ that satisfies $\tau_1 \cdot \tau_2$ (resp. $\tau_1 \vee \tau_2$) and
   $NewC = merge(C_2, C)$;

iii. $DecOne(\tau_1 \wedge \tau_2, OldC, NewC)$
   (resp. $DecOne(\tau_1 | \tau_2, OldC, NewC)$) iff
   $\exists\, C_1$ s.t. $DecOne(\tau_1, OldC, C_1)$,
   $\exists\, C_2$ s.t. $DecOne(\tau_2, C_1, NewC)$.

Since $\wedge$ and $|$ do not generate critical points, *DecOne* is just called onto the first branch and the resulting constraint store is passed to the call on the

second branch (rule iii); the definition on trace expressions whose outmost operator is either $\cdot$ or $\vee$ is more complex as a further merge with the constraint store generated by these operators is required (rule ii). We recall that $C\tau_0$ is the initial constraint store: it contains one set $\{ag\}$ for each agent ag involved in τ.

Definition 11 (Monitoring Safety (MS)). *A partition of Ags P is Monitoring Safe either if DecOne(τ, $C\tau_0$, P) holds, or if DecOne(τ, $C\tau_0$, P′) holds and P can be obtained from P′ by aggregating some groups in it.*

We are just one step away from giving the *DecAMon* definition: we need to introduce the *findall*(Var, *Goal*, *Res*) extra-logical predicate which creates a list *Res* of Var instances obtained by backtracking over *Goal*. We are ready:

$$DecAMon(\tau) = MSs$$
$$\text{iff } findall(P, DecOne(\tau, C\tau_0, P), MSs)$$

Lemma 1. *Let τ be a trace expression, let (e_1, e_2) be a critical point generated by a sub-expression of τ, and let τ_{prj} be a trace expression obtained from τ by projection (namely, $\tau_{prj} = \Pi(\tau, GrI)$ for some group of agents $GrI \subseteq Ags$).*

M_{GrI} detects the uniqueness or sequentiality violation of (e_1, e_2)

$$\Longleftrightarrow$$

the trace expression τ_{prj} checked by M_{GrI} (the monitor for GrI) contains a sub-expression that generates a critical point (e_1, e_2).

Proof. $\Longrightarrow$
We first demonstrate that the trace expression τ_{prj} checked by M_{GrI} contains both e_1 and e_2.

M_{GrI} detects the uniqueness violation of (e_1, e_2) in the current state of the protocol represented by τ_c if either e_1 takes place, τ_c can move to τ'_c ($\tau_c \xrightarrow{e_1} \tau'_c$), and there is no state τ''_c that can be reached by repeatedly applying the $\rightarrow$ relation starting from τ'_c s.t. $\tau''_c \xrightarrow{e_2} \tau_{next}$, for some τ_{next}, or viceversa (e_2 takes place moving the protocol to a state where e_1 cannot "fire" any transition, and from which there are no reacheable states where e_1 can fire any transition).
In order for M_{GrI} to detect the uniquennes violation in both cases above, τ_c must contain both e_1 and e_2. Since τ_c is a sub-expression of τ_{prj}, τ_{prj} must contain both e_1 and e_2.

M_{GrI} detects the sequentiality violation of (e_1, e_2) in the current state of the protocol represented by τ_c if either e_2 takes place and there is no τ_{next} s.t. $\tau_c \xrightarrow{e_2} \tau_{next}$ (when e_2 takes place, it was not foreseen by the protocol), or if e_1 takes place, $\tau_c \xrightarrow{e_1} \tau'_c$, and an event $e_3 \neq e_2$ takes place. Since the protocol τ'_c can move only if e_2 takes place because of the sequentiality between e_1 and e_2, no transition is possible from τ'_c and a violation is detected by M_{GrI}.
In order for M_{GrI} to detect the violation in both cases above, τ_{prj} must contain both e_1 and e_2.

We can complete the " $\Longrightarrow$ " proof: since τ_{prj} contains both e_1 and e_2 and they generated a critical point (e_1, e_2) in some sub-expression of τ, they must generate a critical point (e_1, e_2) also in some sub-expression of τ_{prj}, as τ_{prj} is obtained from τ by projection. □

Proof. $\Longleftarrow$

By definition of critical point: the sub-expression τ_{prjSub} of τ_{prj} generates a critical point (e_1, e_2) iff

- $\tau_{prjSub} = \tau_1 \vee \tau_2$ and $\exists e_1 \in \mathit{first}(\tau_1)$, $\exists e_2 \in \mathit{first}(\tau_2)$ s.t. $involved(e_1) \cap involved(e_2) = \emptyset$, or
- $\tau_{prjSub} = e_1{:}\tau_1$ and $\exists e_2 \in \mathit{first}(\tau_1)$ s.t. $involved(e_1) \cap involved(e_2) = \emptyset$, or
- $\tau_{prjSub} = \tau_1 \cdot \tau_2$ and $\exists e_1 \in last(\tau_1)$, $\exists e_2 \in \mathit{first}(\tau_2)$ s.t. $involved(e_1) \cap involved(e_2) = \emptyset$.

If $\tau_{prjSub} = \tau_1 \vee \tau_2$ and both e_1 and e_2 take place, the first event taking place leads to a protocol state where the second event cannot fire any transition (we remind the assumption that all events are distinct): M_{GrI} detects a violation.

If $\tau_{prjSub} = e_1{:}\tau_1$ and e_2 takes place in τ_{prjSub}, τ_{prjSub} cannot move to a next state since e_2 is not an expected event in τ_{prjSub}. If e_1 takes place, then $\tau_{prjSub} \xrightarrow{e_1} \tau_1$ but if an event e_3 different from e_2 takes place immediately after, then there is no τ_{next} s.t. $\tau_1 \xrightarrow{e_3} \tau_{next}$. In both cases, M_{GrI} detects a violation.

If $\tau_{prjSub} = \tau_1 \cdot \tau_2$ and e_2 takes place in τ_{prjSub}, τ_{prjSub} cannot move to any next state. If e_1 takes place, then $\tau_{prjSub} \xrightarrow{e_1} \tau_2$ but, if an event e_3 different from e_2 takes place immediately after, there is no τ_{next} s.t. $\tau_2 \xrightarrow{e_3} \tau_{next}$. In both cases, M_{GrI} detects a violation.

□

Lemma 2. *Let τ be a trace expression, let (e_1, e_2) be a critical point generated by a sub-expression of τ.*

M_{Ags} *detects the uniqueness or sequentiality violation of* (e_1, e_2)

$$\Longleftrightarrow$$

the trace expression τ checked by M_{Ags} contains a sub-expression that generates a critical point (e_1, e_2).

Proof. By lemma 1, when $GrI = Ags$ and $\tau = \Pi(\tau, Ags)$.

□

We say that a monitor M "checks" a trace expression τ if M is in charge for verifying that the events it observes do not violate the current state of the protocol, and the initial state of the protocol is represented by τ. Theorem 3 demonstrates that a partition P computed by *DecOne* is monitoring safe.

Theorem 3. *Let τ be a trace expression involving agents Ags, let $C\tau_0$ be τ initial constraint store, let $P = \{Gr1, Gr2, \dots, GrN\}$ be one partition computed by DecOne(τ, $C\tau_0$, P), and let (e_1, e_2) be a critical point generated by a sub-expression τ_{sub} of τ.*

The centralized monitor M_{Ags} that checks τ detects a violation of (e_1, e_2) $\iff$ there exists M_{GrI} that checks $\Pi(\tau, GrI)$ which detects a violation of (e_1, e_2).

Proof. $\Longrightarrow$
Since (e_1, e_2) is a critical point generated by some sub-expression of τ, for the definition of *DecOne* there must be one group of agents $GrI \in P$ that contains both ag_1 and ag_2 s.t. $ag_1 \in involved(e_1)$ and $ag_2 \in involved(e_2)$. The projection functioning ensures that all the events that involve one agent in *GrI* are kept in $\Pi(\tau, GrI)$. Thus, the trace expression $\Pi(\tau, GrI)$ will cointain both e_1 and e_2 as *GrI* contains both ag_1 and ag_2, and (e_1, e_2) is a critical point generated by some sub-expression of $\Pi(\tau, GrI)$.

M_{GrI} can detect the violation of (e_1, e_2) for Lemma 1.

$\Longleftarrow$
(Reductio ad absurdum) Let us suppose that M_{GrI} detects a violation of (e_1, e_2). If M_{Ags} does not detect that violation, then, for Lemma 2, the trace expression τ checked by M_{Args} does not contain any sub-expression that generates (e_1, e_2). This contradicts the theorem hypothesis that (e_1, e_2) is a critical point generated by a sub-expression τ_{sub} of τ.

□

Theorem 3 answers the research question addressed by this chapter: "how to ensure that the system made up of the decentralized monitors detects all and only the same protocol violations that a single centralized monitor observing the MAS would detect."

9.5 Implementation and Experiments

DecAMon has been implemented in SWI-Prolog. The code amounts to almost 600 lines. The choice of Prolog was due to many reasons: one-to-one correspondence between the transition and empty rules definitions and their rule-based implementation; built-in support to cyclic terms and to the recognition that a cyclic term has already been met; built-in support to backtracking over goals; availability of Prolog-based tools for trace expressions management.

When the protocol has as many critical points as AIP_5, computing all the MSs may require too much time. *DecOne* can be used instead of *DecAMon* to compute one MS at a time. As an example, calling *DecOne* of AIP_5 produced the first result in 9 ms, the second one in 8 ms, the third in 1 ms. Although *DecOne* might not return the best MS according to the designer or the runtime environment needs, its result is guaranteed to be monitoring safe.

If time is not an issue, however, all the MSs can be generated for further post-processing. We implemented the following functionalities which operate on a set of monitoring safe partitions:

1. removing non minimal partitions from the set;
2. selecting those partitions that contain N agents groups or less (resp. more), where N is given;
3. selecting those partitions that contain M singleton agents groups or less (resp. more), where M is given;
4. selecting those partitions where the agents in the set D, given as input in form of a Prolog list, are all disjoint;
5. selecting those partitions where the agents in the set T, given as input in form of a Prolog list, are all together.

We run experiments with the protocols introduced in the previous sections, AIP_1 to AIP_7, plus the following four. We used a MacBook Pro (Retina, 13-inch, Early 2015) with Processor 2,7 GHz Intel Core i5, Memory 8 GB 1867 MHz DDR3, SWI-Prolog version 7.2.3.

$$\begin{aligned}\mathrm{AIP}_8 = (&alice \overset{submit}{\Longrightarrow} aamas : aamas \overset{ack}{\Longrightarrow} alice : \epsilon) \mid \\ &(bob \overset{submit}{\Longrightarrow} aamas : aamas \overset{ack}{\Longrightarrow} bob : \epsilon) \mid \\ &(carol \overset{submit}{\Longrightarrow} aamas : aamas \overset{ack}{\Longrightarrow} carol : \epsilon)\end{aligned}$$

respects the connectedness for sequence condition: the agents can be monitored independently.

$$\begin{aligned}\mathrm{AIP}_9 = (&alice \overset{submit}{\Longrightarrow} aamas : chair \overset{review}{\Longrightarrow} bob : \\ &(aamas \overset{accept}{\Longrightarrow} alice : \epsilon) \vee (aamas \overset{reject}{\Longrightarrow} alice : \epsilon)) \\ \mid (&bob \overset{submit}{\Longrightarrow} aamas : chair \overset{review}{\Longrightarrow} alice : \\ &(aamas \overset{accept}{\Longrightarrow} bob : \epsilon) \vee (aamas \overset{reject}{\Longrightarrow} bob : \epsilon))\end{aligned}$$

demonstrates that *DecAMon* may return non minimal monitoring safe partitions, which are detected during the post-processing stage. The two only MMSs are $\{\{chair, aamas\}, \{alice\}, \{bob\}\}$ and $\{\{alice, bob\}, \{chair\}, \{aamas\}\}$ but *DecAMon* also returns $\{\{aamas, alice, bob\}, \{chair\}\}$, besides others, where AAMAS is uselessly grouped with Alice and Bob.

The other two protocols are variants of the Alternating Bit Protocol (ABP) described in (Deniélou and Yoshida, 2012). The ABP is an infinite iteration, where the following constraints have to be satisfied for all occurrences of the interactions:

– The n-th occurrence of message `m1` must precede the n-th occurrence of `m2` which in turn must precede the n-th occurrence of `m3`.

– For $k \in \{1, 2, 3\}$, the n-th occurrence of mk must precede the n-th occurrence of the acknowledge ak, which, in turn, must precede the $(n + 1)$-th occurrence of mk.

Because of space constraints, we do not show here the trace expressions corresponding to ABP_{norm} and ABP_{crit}. The difference between them is that in ABP_{norm}, `m1` $= bob \overset{m1}{\Longrightarrow} alice$, `m2` $= bob \overset{m2}{\Longrightarrow} carol$, `m3` $= bob \overset{m3}{\Longrightarrow}$, and the acknowledges flow in the opposite direction: $\mathrm{M}_{\{bob\}}$ can monitor all the protocol, as Bob is involved in all the interactions.

Protocol	*Dec* **(ms)**	*Dec* **(♯)**	**MMS (ms)**	**MMS (♯)**
aip1	2	4	2	4
aip2	1	4	1	4
aip3	1	4	1	4
aip4	1	1	1	1
aip5	122400	5632	74711	5632
aip6	308	256	139	256
aip7	309	128	37	128
aip8	1	1	1	1
aip9	1	16	1	2
abp_norm	14	1	1	1
abp_crit	8	16	1	16

DecAMon execution time (*Dec* **(ms)**);
MSs returned by *DecAMon* (*Dec* **(♯)**);
execution time of the tool for removing non-minimal MSs (**MMS (ms)**);
computed MMSs (**MMS (♯)**).

TABLE 9.1. Experimental results: using DecAMon to extract minimal monitoring safe partitions.

In ABP$_{crit}$, instead, `m1` = $alice \overset{m1}{\Longrightarrow} bob$, `m2` = $carol \overset{m2}{\Longrightarrow}$, `m3` = $emma \overset{m3}{\Longrightarrow} frank$ (with their respective acknowledges), so the connectedness for sequence of `m1`, `m2`, and `m3` cannot be guaranteed by one monitor alone.

For each protocol we measured the time required by *DecAMon* to compute its output, the number of MSs computed by *DecAMon*, the time required to remove the non minimal partitions from *DecAMon* output, and the number of MMSs. W.r.t. Table 9.1, we highlight the following aspects:

– the number of computed MSs depends on the trace expression structure and not on its length: AIP$_6$ and AIP$_7$ only differ for one operator, but they give different results;

– the number of computed MSs of AIP$_4$, AIP$_8$, ABP$_{norm}$ is 1: this means that the monitoring can be fully decentralized, as *DecAMon* returns only the partition with one singleton group for each agent;

– the number of computed MSs of AIP$_9$ is different from the number of MMSs: *DecAMon* may return non minimal partitions.

Since the more groups in the MMS, the better from the decentralization point of view, selecting a MMS with a high number of groups is a good choice for decentralizing as much as possible. Another criterion for preferring a MMS w.r.t. another could be the number of singleton groups, which correspond to agents that can be monitored on their own. Table 9.2 shows the results of other post-processing functions, namely the number of MMSs that contain at least 1, 5, 7, 9 agents groups for each protocol, and the number of MMSs that contain at least 1, 5, 7, 9 singleton groups. Although Table 9.2 only reports numbers, the post-processing tools return all the partitions that meet the given conditions. The MAS designer or a software agent in charge for the dynamic

Protocol	$\geq 1g$	$\geq 5g$	$\geq 7g$	$\geq 9g$	$\geq 1s$	$\geq 5s$	$\geq 7s$	$\geq 9s$
aip1	4	0	0	0	4	0	0	0
aip2	4	0	0	0	4	0	0	0
aip3	4	0	0	0	4	0	0	0
aip4	1	0	0	0	1	0	0	0
aip5	5632	5632	1600	64	5632	1600	64	64
aip6	256	256	128	0	256	128	128	0
aip7	128	128	128	0	128	128	128	0
aip8	1	0	0	0	1	0	0	0
aip9	2	0	0	0	2	0	0	0
abp_norm	1	0	0	0	1	0	0	0
abp_crit	16	0	0	0	16	0	0	0

Number of MMSs that contain at least 1, 5, 7, 9 agents groups (columns $\geq 1g$, $\geq 5g$, $\geq 7g$, $\geq 9g$).
Number of MMSs that contain at least 1, 5, 7, 9 singleton groups (columns $\geq 1s$, $\geq 5s$, $\geq 7s$, $\geq 9s$).

Table 9.2. Experimental results: filtering minimal monitoring safe partitions using different post-processing functions.

reconfiguration of the MAS monitoring activity can select one among them and can impose further conditions such as having some agents disjoint or together. By running this tool we discovered for example that there is no MMS of AIP_5 where b, c, d are together, and there are 4224 MMSs where b and m are disjoint.

9.6 Discussion

The literature on Distributed Runtime Verification (DRV) is still very limited.

BSPL (Chapter 5) supports a rich variety of practical protocols and can be realized in a distributed asynchronous architecture where the participating agents act based on local knowledge alone; in this way DRV of declarative protocols is naturally supported. The major difference of our work in comparison to BSPL, is that we face the challenge of DRV of those protocols that do not satisfy the unique point of choice and connectedness for sequence conditions.

Testerink et al. (Testerink, Bulling, and Dastani, 2016; Testerink, Dastani, and Bulling, 2016) present a formal model for decentralized monitors that supports their formal analysis to face the robustness and security, and a theoretical analysis of distributed runtime norm enforcement. They synthesize the properties that each local monitor is able to verify, expressed in LTL, in order to build a consistent representation of the global state of the world. We do the opposite: we start from a global protocol modeling how the world should behave, and create sub-protocols that involve disjoint groups of agents, in such a way that violations to the global protocol can be discovered by at least one of the monitors in charge for these groups.

The work (Mostafa and Bonakdarpour, 2015) which is closer to ours ad-

dresses the following problem: "given a distributed program D and an LTL_3 property ϕ, construct a set of monitor processes whose composition with D can evaluate ϕ at runtime in a sound, complete, and decentralized fashion." The main differences with our proposal consist in the observed events, which are related to the execution of a program and not to communicative behavior in a MAS, and, most importantly, the use of LTL_3 for specifying system properties; in Chapter 4 we have shown that trace expressions are strictly more expressive than LTL_3 when used for runtime verification. Falcone et al. (Falcone, Cornebize, and Fernandez, 2014) propose an efficient and generalized decentralized monitoring algorithm to detect violation of any regular specification by local monitors without central observation point; also in this case the main difference with our work is the expressive power of the employed formalism for specifications.

Decentralizing the runtime monitoring using *DecAMon* can prove useful in many situations. The applications we are actually looking at fall in the e-health and well-being domains that we started exploring in the last year (Aielli et al., 2016; Ferrando, Ancona, and Mascardi, 2016). If we have a global protocol describing the expected behavior of a system of communicating low-power wearable devices able to measure vital parameters to check the health conditions of a person, we would like to add lightweight monitors on top of them to monitor only those events "local" to the devices, still being sure that global protocol violations will be detected. In these scenarios, proximity of the monitor to the device is of paramount importance.

For what concerns the time complexity of computing a Minimal Monitoring Safe partition, we suspect that the problem can be reduced to computing a solution to a Minimal Constraint Network (Montanari, 1974), recently proven to be NP-hard (Gottlob, 2012). For the applications we have in mind, the need for decentralizing a protocol for monitoring purposes arises only seldom, so the (possibly) high complexity of *DecAMon* can be tolerated. Experiments on several protocols have empirically shown that, once the protocol has been decentralized, the time complexity of monitoring is linear in the trace length, and does not depend on the number of involved agents.

10 *Decentralized Runtime Verification of Agent Interaction Protocols with Gaps*

"Whenever people agree with me
I always feel I must be wrong."
- Oscar Wilde

The work presented in this chapter represents the bridge between Chapter 7 and Chapter 9; here we take advantage of the decentralized approach also in presence of uncertainty on the interactions observed by the distributed monitors. In Chapter 7, we presented Probabilistic Trace Expressions (PTEs), the extension of the trace expression formalism with probabilities. Thanks to this extension, we can achieve the runtime verification of systems also with the presence of gaps inside the analyzed traces. Since one of the main focuses in the thesis is to use trace expressions to achieve the runtime verification of AIPs, it is natural to show how we can use this extended formalism also to manipulate them. As we presented in Chapter 9, in order to dynamically verify the behavior of MAS, which are complex systems, a decentralized solution able to scale with the number of agents is necessary. When, for physical, infrastructural, or legal reasons, the monitor is not able to observe all the events emitted by the MAS, gaps are generated. In this chapter we present a runtime verification decentralized approach to handle observation gaps in a MAS using PTEs.

10.1 Introduction

As we presented in Chapter 9, Distributed Runtime Verification is a relatively new research sub-field[1] aimed at designing fault-tolerant distributed algorithms that monitor other distributed algorithms, with the end goal of developing lightweight software systems that are more efficient that traditional verification techniques (Bonakdarpour et al., 2016a,b). The literature on DRV is almost limited (Bartocci, 2013; Falcone, Cornebize, and Fernandez, 2014; Fraigniaud, Rajsbaum, and Travers, 2014; Fraigniaud et al., 2014; Herlihy, 1991; Mostafa and Bonakdarpour, 2015) and becomes even more limited when we consider DRV of a special kind of systems: multiagent systems. In the MAS area, in fact, we are only aware of our own previous works presented in Chapter 8 and Chapter 9.

This chapter addresses the two issues above, decentralized runtime verification of partially observable systems, in a MAS context. The findings presented in this work can be generalized and applied to other kinds of systems, but – for presentation purposes – we concentrate our investigation on MAS.

In offline RV gaps in the trace logs are due to the process of sampling observed events in order to reduce the monitoring overhead. Gaps can also be met in online RV, where the system behavior is analyzed while the system is running and problems with the infrastructure, privacy and legal issues that prevent the monitor to observe some kind of events, faults in the monitor observation capabilities, may generate gaps. For instance, we may be interested in checking communication protocols, where the observed events are messages and the properties we want to check are protocols. In such kind of scenarios, we can still have gaps, but, instead of having them for optimizations (sampling) like for the offline runtime verification process, we may have gaps due to issues with the network, lost of messages, and so on. Although the problems raised by online and offline RV with gaps share many similarities, the online setting is much more challenging. Each time a gap is perceived, the monitor must make guesses on the possible actual events that the gap represents and save all the states generated by these guesses. A possibly huge logical tree-like structure with states as nodes, and moves from states to states as edges, represents the open possibilities[2]. In offline RV, this logical tree-like structure can be explored following a depth-first search, requiring a limited amount of memory. If the RV takes place online, its exploration must follow a breadth-first strategy, with much more space needed to save the states, as the final trace of events is unknown and the levels of the structure are generated and explored at the same time. In order to cope with the state space explosion due to guesses in

[1] The First Workshop on DRV was held at Bertinoro in May 2016, `http://www.labri.fr/perso/travers/DRV2016/`, and the first survey has been published in October 2016 (Bonakdarpour et al., 2016b). It references 18 papers only and many of them, such as (Fraigniaud, Rajsbaum, and Travers, 2014; Fraigniaud et al., 2014; Herlihy, 1991), deal with issues which fall outside the scope of our investigation.

[2] In the remainder we will use the term "branch" to denote paths in this logical structure, and we will sometime use "states" meaning "the final states of all the possible branches", when this does not generate confusion.

the online RV scenario, we propose to *decentralize* the monitoring activity.

RV decentralization is a very natural choice when the system under monitoring is a MAS, which is *distributed by definition*, and may improve *efficiency*, as the verification process can be spread on different machines improving performance; *scalability*, as under some conditions depending on the protocol (see Chapter 9) it is possible to associate one monitor with each agent in the MAS, keeping under control the RV complexity even when the number of agents grows; *feasibility*, as for physical/logical/legal reasons one single monitor might not be able to observe all the events generated by the MAS.

The feature that is usually subject to verification (both static and dynamic) in a MAS is its *communicative behavior* (Baldoni, Baroglio, and Capuzzimati, 2014a; Baldoni et al., 2015; Chopra, Christie, and Singh, 2017; Chopra and Singh, 2015a; Singh, 2011a; Winikoff, Liu, and Harland, 2004b; Yadav, Padgham, and Winikoff, 2015b; Yolum and Singh, 2002). With respect to (Stoller et al., 2011) and Chapter 7, in this chapter we do not aim at verifying temporal properties. Rather, we want to check the conformance of the MAS actual communicative behavior to an AIP that models the allowed interactions among agents, under the hypotesis that some interactions could not be observed. The research question we address is thus

how to evaluate the probability that a MAS satisfies an AIP,
in the presence of gaps?

In Chapter 7 we introduced Probabilistic Trace Expressions (PTEs) and the theory behind them. In this work we take a more pragmatical perspective and we show how to use PTEs for decentralized RV of AIPs within MAS with gaps[3].

10.2 Exploiting DecAMon for PTEs

In Chapter 9 we presented the DecAMon algorithm to decentralize agent interaction protocols modeled using trace expressions. There, we defined the notion of “monitoring safe” partition (Definition 11). A partition can be used to drive the distribution process. To decentralize the monitoring activity, we project the global AIP onto each subset of agents belonging to the partition, where by “projection” we mean that we maintain only the interactions involving agents in the chosen subset (Section 8.2). In general, not all the partitions can be used for the RV decentralization. A partition that can be used to decentralize the RV of a protocol is called “monitoring safe” and the algorithm presented in Chapter 9 generates all the monitoring safe partitions for a given AIP.

Since under the conditions considered in this chapter we may observe gaps, we could not have only one single state representing the current situation of the protocol, like it happens in our previous works; instead, we have to maintain all the states that may be possibly reached “via the gaps”. As already

[3] In a certain way, this chapter can be seen as the bridge between the approach presented in Chapter 9 and the formalism extension presented in Chapter 7.

introduced in Chapter 7, each state can be represented as a tuple $\langle \tau, \pi, evs \rangle$, where τ is the PTE representing the current state of the protocol and π is the joint probability that the sequence of events evs is compliant with τ.

Let us name M_0 the set of possible initial states of the monitor (as there may be more than one). The number 0 stands for the $0th$ iteration, since at the beginning we have not consumed any event yet. We can first run DecAMon on the global AIP to find a good set of monitoring safe partitions and, after that, we can use one of them to project the τs in M_0 onto the subsets of the agents. Once we have obtained the distributed versions of the initial τs via projection, we can generate one monitor for each partition, and decentralize the RV.

The combination of decentralization and lack of information calls for a synchronized management of gaps. Since each monitor has a different state representing its current protocol evolution, when there is an observation gap, each monitor can have different opinions about which are the correct events that might suitably "fill the gap". The local perspectives can be compared and used by the monitors to cut wrong guesses, and hence wrong states, on the basis of distributed knowledge. Despite the overhead due to synchronization, this approach may dramatically improve performance, as discussed in the next sections.

10.3 Handling Gaps in Decentralized RV

Gaps represent lack of information, thus a point (or points) in the event trace where the monitor does not know what event had been actually generated by the system under monitoring. In the remainder we will write that "gaps can be observed", in the sense that a monitor can realize that something went wrong and that an event was generated by the system, and not correctly observed. We also assume that, in a decentralized setting, when one monitor "observes a gap", all the monitors "observe a gap" as well. From a technical viewpoint, this could be obtained by forcing one monitor to inform the others when it observes a gap. This would require some shared clock among the monitors as, in order for our algorithm to work, the gap must take place at the same time for all the monitors hence raising clock synchronization issues. Given that these issues are well known and well studied in distributed systems (Lamport, 1978), we leave them out of our investigation.

When a centralized monitor observes a gap, since it is the only monitor checking the event trace w.r.t. the AIP specification, it can make guesses on what the gap is and reason on its own guesses, eventually tagging some of them as wrong due to successive observations. When there are many monitors, each one monitoring a subset of the agents, and hence a sub-protocol of the global AIP, each monitor can still suppose what the observed gap is, but the reasoning on its suppositions must be shared with the others. This sharing phase among the monitors is crucial, because it allows them to cut wrong branches on the basis of what other monitors suppose, or what they are fully sure of.

Let us consider two monitors m_1 and m_2 that observe a gap. Given that the protocols driving the two monitors are different, although being derived via projection from the same global protocol, m_1 might suppose that the events admissible for filling the observed gap are e_1 and e_2, while m_2 could instead suppose that admissible events are e_2 and e_3. Both m_1 and m_2 must keep track of these possibilities in their local knowledge bases, and – so far – they do not need to share they guesses.

Let us now suppose that in the current state of m_1, in the branch where e_1 was supposed to have taken place, the only successive possible event is e_4, while in the branch for e_2 the only possible event is e_5. If, after the gap, m_1 observes e_5, it can cut the branch where the gap was associated with e_1, because e_5 would not be allowed after e_1. The gap before e_5, that could be filled in principle by e_1 and e_2, becomes bound - "without any doubt"[4] - to e_2. After having found the right value for the gap and cut one branch, m_1 informs m_2 allowing it to cut the branch where the value for that gap was guessed to be e_3. In this way, both m_1 and m_2 can continue the verification process supposing that the unobserved event represented by the gap was e_2, with some given probability due to the probability associated with e_2 in the PTE modelling the protocol.

Before presenting the decentralized monitoring algorithm, we make some considerations on the kind of gaps a monitor can observe. So far, we considered generic events. This is correct and consistent with the general approach presented in Chapter 7, but in a MAS scenario where PTEs model agent interaction protocols we can be more specific. In this scenario, in fact, the universe of events is *Msgs*, namely the universe of the possible messages among agents. Such special events can be represented as $a_1 \stackrel{c}{\Longrightarrow} a_2$ (as we have already done in the rest of the part), meaning that agent a_1 sends a message to a_2 with content c. Since messages are composed by (at least) three mandatory components, sender, receiver and content, there can be many partially instantiated gaps such as:

- $gap(a_1 \stackrel{_}{\Longrightarrow} a_2)$, where the content of the message is unknown;
- $gap(_ \stackrel{m}{\Longrightarrow} a_2)$, where the sender is unknown;
- $gap(a_1 \stackrel{m}{\Longrightarrow} _)$, where the receiver is unknown.

Although, for sake of clarity, in the sequel we consider gaps where neither the sender, nor the receiver, nor the content are known (total absence of information), all the combinations of "information holes" are possible, and

[4] Modulo the assumption that observed events are compliant with the foreseen protocol. Gaps may inevitably generate false negatives. In this case, m_1 assumes that the gap was e_2 because this would be consistent with the successive observation of e_5 and with the protocol to be respected. If the gap were any other event, a protocol violation would have taken place and m_1 should have raised a protocol monitoring exception. Depending on the protocol, the violation could be recognized later on, or never. Suppose for example an infinite protocol where only *a*s are allowed. A gap will be necessarily filled with *a* even if the actual event was *b*, and if the successive observed events are all *a*s, the violation will never be discovered.

partially instantiated gaps may be exploited to reduce branches due to guesses. The algorithm presented in the next section can be easily adjusted to take partially instantiated gaps into account.

10.3.1 Synchronizing Decentralized Gaps Management

We present the algorithm used by the decentralized monitors to synchronize the gaps management, in order to cut useless branches and check the compliance of interactions with the protocol. When an event is generated by the system, two different situations can take place.

Case 1: The event is not a gap

If the event is not a gap, each monitor that observed it can use the event for updating its local state(s). If some branches have been removed as in the previous example involving m_1 and m_2, the monitor has to inform the other monitors of the associations between gaps and events that are not admissible any longer. This phase can be reiterated until all the monitors have cut all the possible wrong brnches, and have nothing more to say. After this synchronizations stage, the monitoring process continues in the normal way.

Case 2: The event is a gap

To keep the presentation simple, we assume that gaps are observed by all the monitors at the same time. Each monitor guesses the events admissible to fill the gap, according to its local states. If the gap is partially instantiated (some of its components were correctly observed, like the sender, or the content, or both), the monitor can use this information to reduce the set of possible candidate events.

The two cases can be seen as a *reduce* and *extend* stages, respectively. When the monitor observes a fully instantiated event it can invalidate zero, one or more branches. If the invalid branches contain gaps, the monitor can also invalidate the associations between these gaps and the guessed events, and can allow the other monitors to invalidate these associations as well via communication. On the other hand, observation of gaps generates as many branches as the events that, according to the AIP, could fill the gap. We can formalize this intuition in the following way.

Given M_0 as the set of global states $\{\langle \tau_1, \pi_1, [] \rangle, ..., \langle \tau_n, \pi_n, [] \rangle\}$.

1. Distribute M_0 with respect to a given partition $P = \{\{ags_1\}, ..., \{ags_{np}\}\}$, projecting the states onto subsets of the agents involved (the function Π projects an AIP τ onto a set of agents ags removing all the events whose sender and receiver do not belong to ags), obtaining

$$M_{0,\{ags_1\}} = \{\langle \Pi(\tau_1, \{ags_1\}), \pi_1, [] \rangle, ..., \langle \Pi(\tau_n, \{ags_1\}), \pi_n, [] \rangle\}$$

$$\dots$$

$$M_{0,\{ags_{np}\}} = \{\langle \Pi(\tau_1, \{ags_{np}\}), \pi_1, [] \rangle, ..., \langle \Pi(\tau_n, \{ags_{np}\}), \pi_n, [] \rangle\}$$

2. Each monitor observes only the event messages involving the agents belonging to its set ags_i:

   a) if the event message is a gap, the monitor guesses what it could be and generates as many states as the possible events (*extend*);

   b) if the event message is ground, the monitor can cut branches, and in this case it communicates with other monitors the gap values that are no longer admissible (*reduce*).

3. If, after observation of an event or because of information received from other monitors, the set of possible current states for a monitor *m* becomes empty, *m* stops the monitoring process, informs all the other monitors, and they also stop monitoring. The absence of possible current states for a monitor is due to a protocol violation that took place, preventing at least one monitor to move a further step. So, the system checked does not satisfy the agent interaction protocol and the associated probability is 0.

4. Else,

   a) if there are no events left to analyze, the monitoring process ends and the resulting probability is evaluated (see after how);

   b) else, repeat from step 2.

To be more clear, in step 2, given the current event message, each monitor queries its current state following the PTE operational semantics presented in Chapter 7 in order to check if the event message is admissible or not. In the updating phase, the monitors inform the others trying to cut not admissible branches.

If the monitoring process ends without violations detected and there are no more events left to analyze, each monitor stops with at least one admissible branch. Each monitor states its own evaluation of the probability that the system's behavior satisfies the agent interaction protocol. This probability can be computed summing up all the joint probabilities contained in all the final states, corresponding to the last nodes of the admissible branches. This leads to having one estimated value for each monitor: we can adopt different strategies to summarize the final, and global, one. One way could be to take the smallest value among all those estimated by all the monitors, meaning that we want to be cautious and we consider the lowest probability of acceptance; otherwise we could take the biggest value, meaning that we want to be optimi stic since we trust the probabilities used in our specification. In other scenarios we could take the means of the values computed by the monitors, or a weighted means where weights model each monitor's trustability, or other domain-dependent strategies.

10.4 Example

We present a simple example helping us to show how the *extend* and *reduce* steps work. We consider a scenario involving a MAS involving four agents:

$\{alice, bob, charlie, dave\}$. The set of events of our interest is the set of messages that these agents can use to communicate with each other.

Given the PTE

$$\tau = \tau_1 \vee \tau_2$$
$$\tau_1 = alice \overset{msg_1}{\Longrightarrow} bob[0.7]{:}(bob \overset{msg_2}{\Longrightarrow} charlie[0.6]{:}\tau_1 | bob \overset{msg_3}{\Longrightarrow} dave[0.4]{:}\epsilon)$$
$$\tau_2 = alice \overset{msg_4}{\Longrightarrow} dave[0.3]{:}(charlie \overset{msg_5}{\Longrightarrow} dave[0.3]{:}\epsilon | bob \overset{msg_3}{\Longrightarrow} dave[0.7]{:}\tau_2)$$

We decentralize τ on each single agent, obtaining[5]:

$$M_{0,\{alice\}} = \{\langle \Pi(\tau, \{alice\}), 1, [] \rangle\} = \{\langle \tau_{alice}, 1, [] \rangle\}$$
$$M_{0,\{bob\}} = \{\langle \Pi(\tau, \{bob\}), 1, [] \rangle\} = \{\langle \tau_{bob}, 1, [] \rangle\}$$
$$M_{0,\{charlie\}} = \{\langle \Pi(\tau, \{charlie\}), 1, [] \rangle\} = \{\langle \tau_{charlie}, 1, [] \rangle\}$$
$$M_{0,\{dave\}} = \{\langle \Pi(\tau, \{dave\}), 1, [] \rangle\} = \{\langle \tau_{dave}, 1, [] \rangle\}$$

where

$$\tau_{alice} = \tau_{1_{alice}} \vee \tau_{2_{alice}}$$
$$\tau_{1_{alice}} = alice \overset{msg_1}{\Longrightarrow} bob[0.7]{:}\tau_{1_{alice}}$$
$$\tau_{2_{alice}} = alice \overset{msg_4}{\Longrightarrow} dave[0.3]{:}\tau_{2_{alice}}$$
$$\tau_{bob} = \tau_{1_{bob}} \vee \tau_{2_{bob}}$$
$$\tau_{1_{bob}} = alice \overset{msg_1}{\Longrightarrow} bob[0.7]{:}(bob \overset{msg_2}{\Longrightarrow} charlie[0.6]{:}\tau_1 | bob \overset{msg_3}{\Longrightarrow} dave[0.4]{:}\epsilon)$$
$$\tau_{2_{bob}} = bob \overset{msg_3}{\Longrightarrow} dave[0.7]{:}\tau_{2_{bob}}$$
$$\tau_{charlie} = \tau_{1_{charlie}} \vee \tau_{2_{charlie}}$$
$$\tau_{1_{charlie}} = bob \overset{msg_2}{\Longrightarrow} charlie[0.6]{:}\tau_{1_{charlie}}$$
$$\tau_{2_{charlie}} = charlie \overset{msg_5}{\Longrightarrow} dave[0.3]{:}\tau_{2_{charlie}}$$
$$\tau_{dave} = \tau_{1_{dave}} \vee \tau_{2_{dave}}$$
$$\tau_{1_{dave}} = bob \overset{msg_3}{\Longrightarrow} dave[0.4]{:}\epsilon$$
$$\tau_{2_{dave}} = alice \overset{msg_4}{\Longrightarrow} dave[0.3]{:}(charlie \overset{msg_5}{\Longrightarrow} dave[0.3]{:}\epsilon | bob \overset{msg_3}{\Longrightarrow} dave[0.7]{:}\tau_{2_{dave}})$$

Let us suppose that the monitors observe a *gap* now. Each monitor moves to a new set of states corresponding to the possible values for the *gap*.

$$M_{0,\{alice\}} \overset{gap}{\rightarrow} \{$$
$$\langle \tau_{1_{alice}}, 0.7, [gap(alice \overset{msg_1}{\Longrightarrow} bob)] \rangle,$$
$$\langle \tau_{2_{alice}}, 0.3, [gap(alice \overset{msg_4}{\Longrightarrow} dave)] \rangle,$$
$$\langle \tau_{alice}, 1, [gap(none)] \rangle$$
$$\} = M_{1,\{alice\}}$$
$$M_{0,\{bob\}} \overset{gap}{\rightarrow} \{$$

[5]The initial probability of each state is 1,
since we do not want to influence the probability evaluation process (multiplication of probabilities).

$$\langle (bob \overset{msg_2}{\Longrightarrow} charlie[0.6]{:}\tau_1 | bob \overset{msg_3}{\Longrightarrow} dave[0.4]{:}\epsilon), 0.7, [gap(alice \overset{msg_1}{\Longrightarrow} bob)] \rangle,$$
$$\langle \tau_{2_{bob}}, 0.7, [gap(bob \overset{msg_3}{\Longrightarrow} dave)] \rangle,$$
$$\langle \tau_{bob}, 1, [gap(none)] \rangle$$
$$\} = M_{1,\{bob\}}$$
$$M_{0,\{charlie\}} \overset{gap}{\rightarrow} \{$$
$$\langle \tau_{1_{charlie}}, 0.6, [gap(bob \overset{msg_2}{\Longrightarrow} charlie)] \rangle,$$
$$\langle \tau_{2_{charlie}}, 0.3, gap(charlie \overset{msg_5}{\Longrightarrow} dave) \rangle,$$
$$\langle \tau_{charlie}, 1, [gap(none)] \rangle$$
$$\} = M_{1,\{charlie\}},$$
$$M_{0,\{dave\}} \overset{gap}{\rightarrow} \{$$
$$\langle \epsilon, 0.4, gap(bob \overset{msg_3}{\Longrightarrow} dave) \rangle,$$
$$\langle (charlie \overset{msg_5}{\Longrightarrow} dave[0.3]{:}\epsilon | bob \overset{msg_3}{\Longrightarrow} dave[0.7]{:}\tau_{2_{dave}}), 0.3, gap(alice \overset{msg_4}{\Longrightarrow} dave) \rangle,$$
$$\langle \tau_{dave}, 1, [gap(none)] \rangle$$
$$\} = M_{1,\{dave\}}$$

Since they observed a *gap*, the monitors do not know what the actual event was. Because of this, they have to generate more branches, where each branch represents a possible value for the gap. This is the *extend* step.

Let us now suppose that the monitors observe event msg_2. Since msg_2 is a ground event, everything is known about it, in particular the monitors know that its sender is *bob* and its receiver is *charlie*. Since the monitors observe only the gaps and the events that involve the agents in the partition they are in charge for, the only monitors that observe msg_2 are $M_{1,\{bob\}}$ and $M_{1,\{charlie\}}$.

By consuming msg_2, the first iteration of the algorithm leads to:

$$M_{1,\{bob\}} \overset{bob \overset{msg_2}{\Longrightarrow} charlie}{\longrightarrow} \{$$
$$\langle \tau_1 | bob \overset{msg_3}{\Longrightarrow} dave[0.4]{:}\epsilon, 0.42, [gap(alice \overset{msg_1}{\Longrightarrow} bob), bob \overset{msg_2}{\Longrightarrow} charlie] \rangle$$
$$\} = M_{2,\{bob\}}$$
$$M_{1,\{charlie\}} \overset{bob \overset{msg_2}{\Longrightarrow} charlie}{\longrightarrow} \{$$
$$\langle \tau_{1_{charlie}}, 0.36, [gap(bob \overset{msg_2}{\Longrightarrow} charlie), bob \overset{msg_2}{\Longrightarrow} charlie] \rangle,$$
$$\langle \tau_{1_{charlie}}, 0.6, [gap(none), bob \overset{msg_2}{\Longrightarrow} charlie] \rangle$$
$$\} = M_{2,\{charlie\}}$$

It is interesting to analyze what happened in $M_{2,\{bob\}}$, where the *reduce* step took place. In fact, the ground event msg_2 makes the other two branches not valid anymore. More in detail, the second branch was $\langle \tau_{2_{bob}}, 0.7, [gap(msg_3)] \rangle$, and $\tau_{2_{bob}}$ does not accept the event msg_2 and cannot move to a new state. In the same way, the PTE in the third branch $\langle \tau_{bob}, 1, gap(none) \rangle$ is τ_{bob}, and τ_{bob} cannot accept the event msg_2 either. Even though this information seems important for monitor $M_{2,\{bob\}}$ only, it is actually of interest also for the other

monitors. In fact, it allows all of them to know "without any doubt" that the only event that can be associated with the first gap is msg_1, since it is the gap value associated with the only possible branch of $M_{2,\{bob\}}$. The monitor $M_{2,\{bob\}}$ can inform the other monitors that the only admissible value for the gap is msg_1. The monitors' new states become:

$$M_{2,\{charlie\}} = \{\langle \tau 1_{charlie}, 0.6, [gap(none), bob \stackrel{msg_2}{\Longrightarrow} charlie] \rangle\}$$

$$M_{1,\{alice\}} = \{\langle \tau 1_{alice}, 0.7, [gap(alice \stackrel{msg_1}{\Longrightarrow} bob)] \rangle\}$$
$$M_{1,\{dave\}} = \{\langle \tau_{dave}, 1, [gap(none)] \rangle\}$$

This example shows how the knowledge of a monitor can have a positive impact on the knowledge of the other monitors. In general, this positive impact can be obtained any time one monitor discovers that one branch is no longer valid and can hence invalidate the associations of events with gaps therein. This information may trigger many communication iterations among the monitors, because, when one monitor is updated it can also "invalidate one branch" and the related gap-events associations, and may need to inform the others of some association which is no longer possible. In the previous example, one single iteration was enough.

As we already anticipated, the proposed approach may lead to false negatives, due to an optimistic approach of the monitors that stubbornly assume that observed events are compliant with the protocol, if there is just one possibility left to make such an assumption. Also in this example, the monitors gave the correctness of the ground event msg_2 (the second event observed) for granted. But let us suppose that the actual event masked by the *gap* was not msg_1, but msg_4, and that the successive message msg_2 was sent from *bob* to *charlie* by mistake and did not comply with the protocol. In this scenario, since the monitors do not know for sure what the first *gap* was, it is reasonable to consider msg_2 a valid message and hence cut the branch where the gap has been supposed to be msg_4. This is a problem intrinsically related to the state estimation approach, since until it is acceptable to observe an event in a state, the monitors keep track of the related branch. Only when a monitor, observing an event, loses all its branches it can conclude that a protocol violation took place because some wrong assumption on gaps – confirmed by successive observations – had been made in the past. This delay in the error detection, which could also be infinite, can be reduced introducing a threshold on the probability that a branch must have to be considered valid. In this way, if after observing an event the probability associated with a branch becomes lower than a chosen threshold, the monitor can cut that branch and make error detection possibly quicker.

10.5 Implementation and Experiments

In our experiments we have considered the four following features:

1. the number of agents involved in the MAS we want to verify at runtime;

Table 10.1. Average time of the centralized and decentralized algorithms; "sh. PTE" stands for "shuffled sub-PTE".

# sh. PTEs	# ag. for sh. PTE	# op. for sh. PTE	Centralized [sec]	Decentralized [sec]
10	10	20	6.64	1.26
10	10	15	8.26	1.04
10	5	20	9.85	1.49
10	5	15	9.92	1.28
10	15	15	14.86	1.23
10	5	10	18.35	1.08
10	15	10	20.25	1.61
10	10	10	29.59	1.98
15	5	15	93.34	2.73
15	15	10	116.61	3.56
10	15	20	126.31	25.32
15	10	10	283.70	4.14
15	5	10	349.30	2.23
20	10	10	355.90	3.99
15	5	20	363.67	5.83
20	5	15	558.59	9.28
20	5	20	801.37	7.82
15	20	10	952.43	12.36
20	5	10	1223.85	10.64
20	15	10	1340.29	9.57
20	20	10	1727.26	2.89

2. the number of *shuffled sub-PTEs* due to shuffle operators | in the AIP: we name shuffled sub-PTE each portion of the PTE composed via a |, so for example $\tau_3 = alice \overset{msg_1}{\Longrightarrow} bob[0.7]{:}\epsilon \mid bob \overset{msg_3}{\Longrightarrow} dave[0.4]{:}\epsilon$ consists of 2 *shuffled sub-PTEs*; we point out that when decentralizing the monitoring, we can associate one different monitor with each shuffled sub-PTE, as shuffled sub-PTE are independent one from the other and can be monitored in a fully decentralized way;
3. the number of operators for each shuffled sub-PTE in the AIP;
4. the number of gaps contained in the analyzed traces.

In Table 10.1, we report the results of our experiments. For each row, we keep the number of shuffled sub-PTE, agents and operators fixed, while we change the length of the traces and the percentage of gaps inside each trace. For each row we executed many different runs and we have measured the total time required for recognizing the set of 300 randomly generated traces. We changed the number of gaps contained inside the traces and we tested both the centralized (Chapter 7) and the decentralized algorithms. In the following, we reported the graphics obtained from such executions.

Concerning the figures, *the traces used in our experiments contain only gaps* (namely, we run experiments in the worst possible scenario), so the algorithm makes only expansions and never reductions. We chose traces with only gaps

to stress the algorithms as much as possible. In real scenarios gaps should be the exceptions, and perfectly observable events the norm.

In Figures 10.1 and 10.2, both the centralized and the decentralized algorithms seem to show linear complexity with respect the number of the agents involved, even if the decentralized algorithm has better performances.

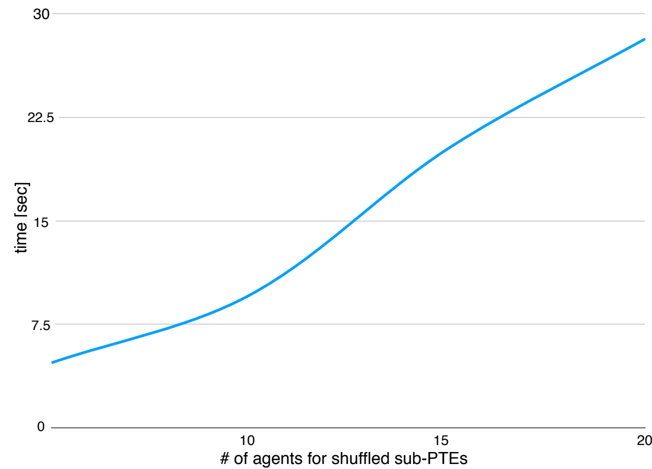

FIGURE 10.1. Centralized algorithm: changing number of agents.

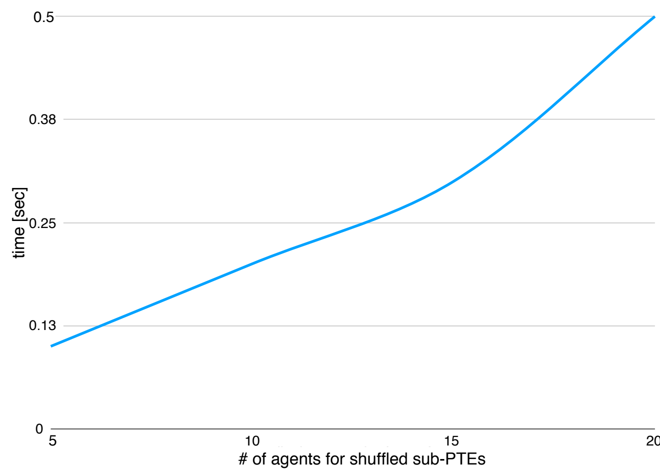

FIGURE 10.2. Decentralized algorithm: changing number of agents.

In Figures 10.3 and 10.4, we can observe that the complexity of the centralized algorithm seems to grow in a quadratic way, while the decentralized one seems to grows linearly. This can be explained by the decentralization of the monitoring of shuffled sub-PTEs, as if we add one operator to each shuffled sub-PTE, the monitor in charge for that shuffled sub-PTE will need to manage one more operator only, whereas the centralized monitor will cope with as many new operators as the shuffled sub-PTEs in the trace expression. We point

out that we use "seems to" to reflect that the complexities emerging from the figures have not been computed on the basis of the algorithm, but have been estimated on the basis of the experiments, and the behaviour in situations involving a limited number of agents, operators, shuffled sub-PTEs, might not be the actual asymptotic behaviour of the algorithm.

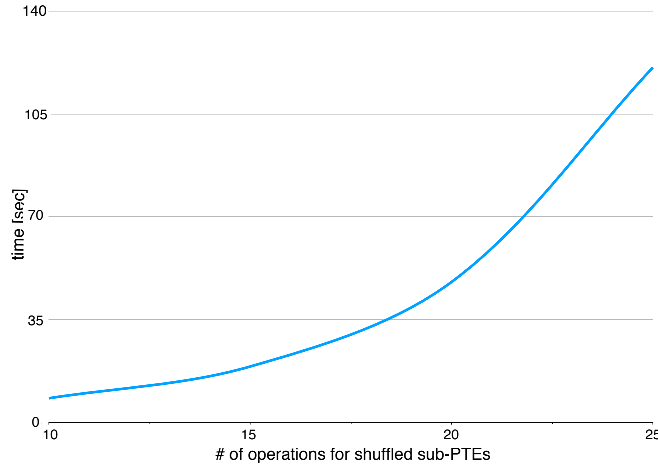

FIGURE 10.3. Centralized algorithm: changing number of operators.

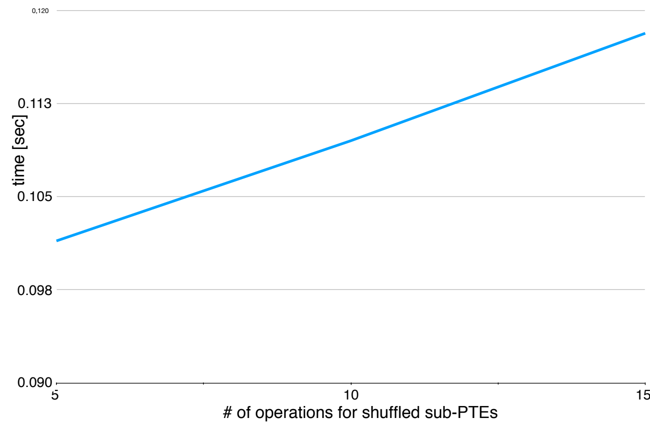

FIGURE 10.4. Decentralized algorithm: changing number of operators.

In Figures 10.5 and 10.6, we can appreciate the real advantages of decentralization, as – from the figures – it seems that we have an exponential complexity for the centralized algorithm and a pseudo-quadratic complexity for the decentralized one. We emphasise that in the decentralized case (Figure 10.6) we were able to run experiments with 40 shuffled sub-PTEs, while in the centralized case we had to stop with half shuffled sub-PTEs, and with an execution time hundred times higher. The number of shuffled sub-PTEs is indeed the feature

which most impacts the algorithms performance, and this in not a surprise; intuitively, when we add a new shuffled sub-PTE we have to interleave it with all the already existent shuffled sub-PTEs. In the centralized case, this brings to a state explosion, while in the decentralized one, since we can decentralize the monitoring of each shuffled sub-PTEs, we simply have to add a new monitor. In this way, we can avoid the state explosion, even if the presence of a new monitor increases the exchange of messages among the monitors needed to synchronize information about gaps.

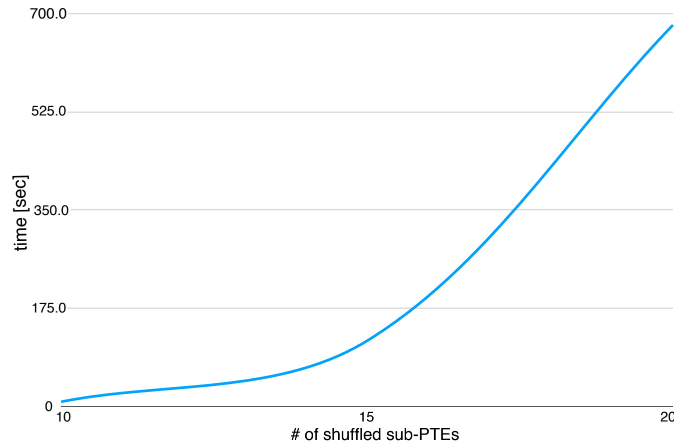

FIGURE 10.5. Centralized algorithm: changing number of shuffled sub-PTEs.

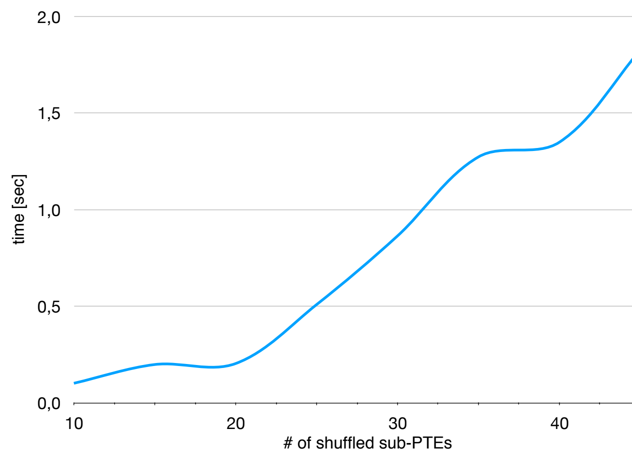

FIGURE 10.6. Decentralized algorithm: changing number of shuffled sub-PTEs.

10.6 Discussion

In this chapter we presented a distributed approach to runtime verification where we may lack some pieces of information about observed events. With respect to standard runtime verification, the state estimation approach allows us to be more reliable, especially in scenarios where partial or total absence of information is frequent.

For the sake of clarity, we considered only totally uninstantiated gaps. This choice has been made to make the development of monitors easier. Naturally, the presence of part of information about the event could be used by the monitors in order to cut useless branches.

11 *Conformance checking*

"I have forced myself to contradict myself
in order to avoid conforming to my own taste."
- Marcel Duchamp

We present an algorithm for establishing a flexible conformance relation between two local agent interaction protocols (LAIPs) based on mappings involving agents and messages, respectively. Conformance is in fact computed "modulo mapping": two LAIPs represented by the corresponding trace expressions τ and τ' may involve different agents and use different syntax for messages, but may still be found to be conformant provided that a given map from entities appearing in τ to corresponding entities in τ' is applied. Since we use the trace expression formalism to model our LAIPs, its expressive power makes the problem of stating if τ conforms to τ' undecidable. We cope with this problem by over-approximating trace expressions that may lead to infinite computations, obtaining a sound but not complete implementation of the proposed conformance check.

The contents of this chapter are published in
(Ancona, Ferrando, and Mascardi, 2018a)

11.1 Introduction

We open the chapter by means of an example. The example allows us to explain the research question we address and how to solve it using the trace expressions formalism.

The example scenario is the following: the company *AI4Tour* develops chatbots interacting with human beings in their daily working activities. *AI4Tour* business is in the tourist sector and chatbots support tourist operators.

A typical conversation between a tourist agency *TourAgency* and the chatbot *TravelChat* starts with the request of whether a plane landed[1], or a cruise ship docked, or a train/bus reached the main city station; the chatbot, by accessing some database or web service in the backend, answers either "yes", "not yet", or "canc" (for canceled), and then becomes available to answer new questions.

The global agent interaction protocol (GAIP) τ which norms the simple multiagent system *mas* involving *TA* (for *TourAgency*) and *TC* (for *TravelChat*) might look like

$$\tau = (TA \overset{landed}{\Longrightarrow} TC{:}\epsilon \ \vee \ TA \overset{docked}{\Longrightarrow} TC{:}\epsilon \ \vee$$
$$TA \overset{train_arrived}{\Longrightarrow} TC{:}\epsilon \ \vee \ TA \overset{bus_arrived}{\Longrightarrow} TC{:}\epsilon) \ \cdot$$
$$(TC \overset{yes}{\Longrightarrow} TA{:}\epsilon \ \vee \ TC \overset{not_yet}{\Longrightarrow} TA{:}\epsilon \ \vee \ TC \overset{canc}{\Longrightarrow} TA{:}\epsilon) \ \cdot \ \tau$$

where *landed* stands for *"did the plane land?"*, *docked* stands for *"did the ship dock?"*, and so on. After receiving one request, the chatbot will react by selecting and sending one answer among the three allowed ones. The protocol definition is recursive: after having received and answered one question, *TravelChat* is ready to start again.

Another company *AI4Moving* develops chatbots that interact with citizens to provide useful information for planning a safe journey within the city boundaries.

A typical conversation between the citizen *C* and the chatbot *MovingChat* starts with *C* asking if some ship docked (because the city traffic is highly impacted by cars and trunks disembarking), or if a train or bus just reached or will reach the main city station (because *C* might consider to take that bus or train, instead of the car); the chatbot answers either "yes", "in one hour", "in two hours", or "not in the next three hours", and moves to the state where it can receive new questions.

The global agent interaction protocol τ' governing *mas′* which involves *C* and *MC* (for *MovingChat*) is

$$\tau' = (C \overset{docked}{\Longrightarrow} MC{:}\epsilon \ \vee \ C \overset{train_in_station}{\Longrightarrow} MC{:}\epsilon \ \vee$$

[1]For sake of clarity, we disregard the facts that a flight is characterized by a code which should be supplied as a parameter to the query, and that when the chatbot answers and becomes ready to manage a new query, it might be able to interact with a travel agency different from *TourAgency*. The trace expressions formalism supports parameters both at the data level (to model messages which only differ for the flight code) and at the agent level (to model multiple concurrent conversations among different agents), but taking parameters into account would make the presentation more complex and we opted for keeping it as simple as possible.

$$C \overset{bus_in_station}{\Longrightarrow} MC{:}\epsilon) \cdot (MC \overset{yes}{\Longrightarrow} C{:}\epsilon \ \vee \ MC \overset{in_1_h}{\Longrightarrow} C{:}\epsilon \ \vee$$
$$MC \overset{in_2_h}{\Longrightarrow} C{:}\epsilon \ \vee \ MC \overset{not_in_3_h}{\Longrightarrow} C{:}\epsilon) \cdot \tau'$$

When the *AI4Tour* company acquires *AI4Moving*, it decides to keep providing the services previously offered by *AI4Moving*, but re-implementing them with its own technologies, in the most efficient and less error-prone way.

W.r.t. to the re-implementation of *MovingChat*, given that *AI4Tour* already developed the *TravelChat* chatbot which clearly shares some similarities with *MovingChat*, the *AI4Tour* software engineers start wondering whether *TravelChat* can be adapted and reused to play the role of *MovingChat*. They address the question: *"can TravelChat safely substitute MovingChat provided that suitable mappings between messages and between agents in mas and mas′ respectively are applied?"*

The intuition behind the "mappings" the *AI4Tour* software engineers are looking for should be clear. We formally define them as a map $M_{\mathcal{M}} : \mathcal{M}_1 \to \mathcal{M}_2$ from the messages that appear in an interaction protocol τ_{ag1} to those that appear in τ'_{ag2}, and a map $M_{\mathcal{A}} : \mathcal{A}_1 \to \mathcal{A}_2$ from the agents that appear in τ_{ag1} to those that appear in τ'_{ag2}, respectively. To answer their "substitutability" question, the engineers must:

(1) Move from the global description τ of how TA and TC interact, to TC's local agent interaction protocol τ_{TC} (LAIP):

$$\tau_{TC} = (\overset{landed}{\Longleftarrow} {}_{TA}{:}\epsilon \ \vee \ \overset{docked}{\Longleftarrow} {}_{TA}{:}\epsilon \ \vee$$
$$\overset{train_arrived}{\Longleftarrow} {}_{TA}{:}\epsilon \ \vee \ \overset{bus_arrived}{\Longleftarrow} {}_{TA}{:}\epsilon) \ \cdot$$
$$(\overset{yes}{\Longrightarrow}{}_{TA}{:}\epsilon \ \vee \ \overset{not_yet}{\Longrightarrow} {}_{TA}{:}\epsilon \ \vee \ \overset{canc}{\Longrightarrow}{}_{TA}{:}\epsilon) \ \cdot \ \tau_{TC}$$

In τ_{TC} we omit to write TC as sender or receiver, as this information is implicit (Section 8.2). Also, if there were messages in τ that involved *TravelChat* neither as the sender nor as the receiver, they would not appear in τ_{TC}.

(2) Move from the global description τ' of how citizens and MC interact, to MC's LAIP, τ'_{MC}:

$$\tau'_{MC} = (\overset{docked}{\Longleftarrow} {}_{C}{:}\epsilon \ \vee \ \overset{train_in_station}{\Longleftarrow} {}_{C}{:}\epsilon \ \vee$$
$$\overset{bus_in_station}{\Longleftarrow} {}_{C}{:}\epsilon) \cdot (\overset{yes}{\Longrightarrow}{}_{C}{:}\epsilon \ \vee \ \overset{in_1_h}{\Longrightarrow}{}_{C}{:}\epsilon \ \vee$$
$$\overset{in_2_h}{\Longrightarrow}{}_{C}{:}\epsilon \ \vee \ \overset{not_in_3_h}{\Longrightarrow} {}_{C}{:}\epsilon) \ \cdot \ \tau'_{MC}$$

(3) Check whether τ_{TC} is conformant to τ'_{MC}; this is achieved by looking for mappings $M_{\mathcal{A}}$ among agents and mappings $M_{\mathcal{M}}$ among messages involved in τ_{TC} and τ'_{MC}, such that *TravelChat* can play the role of *MovingChat* in *mas′*, still ensuring that the GAIP τ' is respected.

(4) Select one couple of mappings among those computed in step (3), $\langle M_{\mathcal{M}}, M_{\mathcal{A}}\rangle$, based on their semantics/pragmatics.

(5) Implement a means to allow *TravelChat* and the citizens to interact, by forcing *TravelChat* to apply the selected mappings when interacting with them.

Agent *TourAgency* in *mas* must be necessarily mapped to *C* in *mas′*. From a semantic and pragmatic point of view, the most reasonable message mapping is the one that maps *docked* ∈ *mas* into *docked* ∈ *mas′* (we abuse notation, and we write *msg* ∈ *mas* to mean that *msg* is one of the messages exchanged by agents belonging to *mas*); *train_arrived* into *train_in_station*; *bus_arrived* into *bus_in_station*; *yes* ∈ *mas* into *yes* ∈ *mas′*; *not_yet* into *in_2_h*; and *canc* into *not_3_h*. The *landed* message is mapped into no message: when "pretending to be *MovingChat*", *TravelChat* will never receive a message whose meaning is close to *landed*, as τ' does not support it. On the other hand, *TravelChat* is not able to discriminate between trains and buses arriving in one or two hours. The mapping of *not_yet* into *in_2_h* is a cautious choice and the citizen will never receive the message *in_1_h*, even if it would be supported by τ'.

From a purely syntactic point of view, and considering protocol specifications only – hence, disregarding the actual services and actions that are triggered by reception of messages –, many other mappings would respect the protocol conformance, including the one that maps *canc* into *yes* ∈ *mas′* and *yes* ∈ *mas* into *not_3_h*.

The research question that we address in this chapter is the one in step (3) above. We point out that such research question cannot be answered by using ontology matching algorithms (Euzenat and Shvaiko, 2007). Ontology matching techniques could indeed be exploited in step (4) of the process, as we discuss in the Conclusions, but not in step (3): an ontology represents static knowledge, not dynamic behaviour. An agent interaction protocol represents dynamic behaviour, not static knowledge. Checking whether a protocol is conformant to another must necessarily take such dynamics into account, which is not required in an ontology matching process and which raises many subtle issues. For example, when moving from τ to τ' to substitute *ag′*, *ag* must be capable to react *at least* to all the "passive events" (for example, receiving a message) that *ag′* can address, and to perform *at most* all the "active events" (for example, sending a message) that *ag′* can perform, at any stage of the protocol. This requirement cannot be satisfied by an ontology matching approach, where it does not even make sense, whereas it is well known in the protocol conformance literature. Depending on the expressiveness of the language used to specify GAIPs, verifying that *ag* can actually substitute *ag′* in a safe way may be more or less complex, or even impossible to perform in an exact way. As an example, recursive protocol definitions are usually disregarded in the literature as they are extremely complex to manage. Since trace expressions, which we use for modeling GAIPs and LAIPs, supports recursion, the existing conformance checking algorithms are not powerful enough for our needs.

Our contribution is an algorithm for addressing step (3) above when GAIPs

are specified as trace expressions. To demonstrate the feasibility of our approach, we present an example implemented in JADE (Bellifemine, Caire, and Greenwood, 2007).

11.2 State of the art

The works closer to our proposal come from Baldoni and Baroglio who, together with their colleagues, introduced the notion of syntactic conformance in the context of interaction protocols for MAS and Service Oriented Computing (SOC) scenarios, starting from 2004. Conformance is based on the notion of interoperability among the entities' policies (e.g. a BPEL process (The OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee, 2007), similar to some extent to our LAIPs) with respect to interaction protocols (e.g. a WS-CDL choreography (Kavantzas et al., 2005), similar to our GAIPs), through the use of finite state automata. While in (Baldoni et al., 2005a; Baldoni et al., 2004, 2005b) protocols were limited to involve two entities only, (Baldoni et al., 2006) presents an extension supporting multiple parties. A further extension is presented in (Baldoni et al., 2009) where decision points are explicitly represented.

Besides the fact that we address the conformance between LAIPs, there are other differences between those works and ours: first, they assume that entities/messages involved in the policy and in the protocol respectively, are exactly the same in order for the conformance check to have some chance to succeed: no notion of mapping is foreseen; second, the expressive power of trace expressions is higher than the expressive power of WS-CDL/BPEL. The presence of expansive subtraces, introduced later on, makes trace expressions able to recognize context-free and non context-free languages, and raises technical problems that do not show up when less expressive formalisms are used.

Among the works by Baldoni and Baroglio's team, however, the most inspiring for our research is (Baldoni, Baroglio, and Capuzzimati, 2014b), recently improved and extended in (Baldoni et al., 2018). That work presents an agent *typing system*, where types are defined as commitments (Yolum and Singh, 2002). The typing includes a notion of compatibility, based on subtyping, which allows for the safe substitution of agents to roles along an interaction that is ruled by a commitment-based protocol. The proposal is implemented in the 2COMM framework (Baldoni, Baroglio, and Capuzzimati, 2013) which is based on the Agent & Artifact meta-model (Omicini, Ricci, and Viroli, 2008), and exploits JADE and CArtAgO (Ricci, Piunti, and Viroli, 2011). Considering the LAIP associated with an agent as its "communicative type" is an almost natural idea in our approach also. The LAIP makes the communicative interface of an agent explicit and can be used both to type check an agent w.r.t. the possibility of entering a MAS normed by some GAIP, and to define a subtyping relation which we name "is conformant to" relation. The main difference between our approach and the one discussed in (Baldoni, Baroglio, and Capuzzimati, 2014b) lies in the adopted formalism and the generality: commitments without

mappings there, trace expressions with mappings here.

Many other works besides those mentioned above aim at defining and testing conformance in the SOC community, including (Bordeaux et al., 2004; Bravetti and Zavattaro, 2007a,b; Busi et al., 2005). None of them uses formalisms which are as powerful as context-free grammars, or more, and none integrates the notion of agents and messages mappings. Also, some of them are limited to two-party protocols.

When moving to the MAS realm, we can devise the same differences between our approach and the others as those identified for SOC approaches: lower expressive power of the adopted formalisms and less generality, due to the absence of mappings in the conformance definition. Among the most notable contributions to protocol conformance, we may mention (Endriss et al., 2003) where Endriss *et al.* identify three levels of conformance, weak, exhaustive, and robust, and explore how a specific class of logic-based agents can exploit an AIP formalism based on simple if-then rules to check conformance a priori or enforce it at runtime. In a similar way, Alberti *et al.* exploit the *S*CIFF abductive proof-procedure (Alberti et al., 2005) for both a priori and runtime verification of compliance of agent interactions (Alberti et al., 2006). In (Chopra and Singh, 2006), Chopra and Singh formalize the notions of conformance, coverage and interoperability. In (Chopra and Singh, 2007) a formal interoperability test for agents is presented. That work considers the presence of two agents only, but in an open scenario where agents can behave differently from the protocol specification. Finally, in (Giordano and Martelli, 2007), Giordano and Martelli address the problem of conformance between an agent and a protocol through an automata-based technique, when the specification of the protocol is given in a temporal action logic.

11.3 LAIP Conformance Modulo Mapping

GAIPS, AGENTS, INTERACTIONS, AND MESSAGES Let *mas* be a multiagent system governed by some GAIP modeled by trace expression τ. We define $GAIP(mas)$ as τ. Let τ be a trace expression involving all and only agents $\mathcal{A}$ and interactions $\mathcal{I}$. We define $AG(\tau)$ as $\mathcal{A}$, $INT(\tau)$ as $\mathcal{I}$, and $MSG(\tau)$ as $\{msg \mid int \in INT(\tau) \text{ and } msg = MSG(int)\}$.

The definitions of AG, INT and MSG hold for both trace expressions and projected trace expressions.

We first give a simpler, but stronger, definition of compliance which does not allow renaming of messages and agents.

Definition 12. *Given two LAIPs τ_{ag1} and τ'_{ag2}, we say that τ_{ag1} is conformant to τ'_{ag2}, written $\tau_{ag1} \leq \tau'_{ag2}$, iff the following conditions are coinductively verified:*

- $\forall msg, ag$ *if* $\exists \tau''_{ag1}$ *s.t.* $\tau_{ag1} \overset{\overset{msg}{\Longrightarrow}_{ag}}{\rightarrow} \tau''_{ag1}$, *then* $\exists \tau'''_{ag2}$ *s.t.* $\tau'_{ag2} \overset{\overset{msg}{\Longrightarrow}_{ag}}{\rightarrow} \tau'''_{ag2} \wedge \tau''_{ag1} \leq \tau'''_{ag2}$;
- $\forall msg, ag$ *if* $\exists \tau'''_{ag2}$ *s.t.* $\tau'_{ag2} \overset{\overset{msg}{\Longleftarrow}_{ag}}{\rightarrow} \tau'''_{ag2}$, *then* $\exists \tau''_{ag1}$ *s.t.* $\tau_{ag1} \overset{\overset{msg}{\Longleftarrow}_{ag}}{\rightarrow} \tau''_{ag1} \wedge \tau''_{ag1} \leq \tau'''_{ag2}$;

- $\{\tau'''_{ag2} \mid \exists msg, ag.\tau'_{ag2} \overset{\overset{msg}{\Longrightarrow}_{ag}}{\rightarrow} \tau'''_{ag2}\} \neq \emptyset$ *implies* $\{\tau''_{ag1} \mid \exists msg, ag.\tau_{ag1} \overset{\overset{msg}{\Longrightarrow}_{ag}}{\rightarrow} \tau''_{ag1}\} \neq \emptyset$.

In the following formalization we assume that $ag1$ is an agent in mas, and $ag2$ an agent in mas', and define $\tau = GAIP(mas)$, $\tau' = GAIP(mas')$, $\tau_{ag1} = \Pi(\tau, ag1)$, $\tau'_{ag2} = \Pi(\tau', ag2)$, $\mathcal{A}_1 = AG(\tau_{ag1})$, $\mathcal{A}_2 = AG(\tau'_{ag2})$, $\mathcal{M}_1 = MSG(\tau_{ag1})$, $\mathcal{M}_2 = MSG(\tau'_{ag2})$.

As introduced in Section 11.1, we consider a map $M_{\mathcal{M}} : \mathcal{M}_1 \rightarrow \mathcal{M}_2$ from the messages that appear in τ_{ag1} to those that appear in τ'_{ag2}, and a map $M_{\mathcal{A}} : \mathcal{A}_1 \rightarrow \mathcal{A}_2$ from the agents that appear in τ_{ag1} to those that appear in τ'_{ag2}.

A more general conformance relation modulo mappings can be defined in terms of the basic conformance relation of Definition 12.

Definition 13. *Given two LAIPs τ_{ag1} and τ'_{ag2}, and two mappings $M_{\mathcal{M}}$ and $M_{\mathcal{A}}$ on messages and agents, respectively, we say that τ_{ag1} is conformant to τ'_{ag2} modulo $M_{\mathcal{M}}$ and $M_{\mathcal{A}}$, written $\tau_{ag1} \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}}\rangle} \tau'_{ag2}$, iff $\langle M_{\mathcal{M}}, M_{\mathcal{A}}\rangle(\tau_{ag1}) \preceq \tau'_{ag2}$.*

With $\langle M_{\mathcal{M}}, M_{\mathcal{A}}\rangle(\tau_{ag1})$ we denote the trace expression obtained from τ_{ag} by replacing all the interactions $\overset{msg}{\Longrightarrow}_{ag}$ and $\overset{msg}{\Longleftarrow}_{ag}$ with $\overset{M_{\mathcal{M}}(msg)}{\Longrightarrow}_{M_{\mathcal{A}}(ag)}$ and $\overset{M_{\mathcal{M}}(msg)}{\Longleftarrow}_{M_{\mathcal{A}}(ag)}$, respectively.

Intuitively, the relation $\tau_{ag1} \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}}\rangle} \tau'_{ag2}$ ensures that $ag1$ can safely substitute $ag2$ in mas', provided that mappings $M_{\mathcal{M}}$ and $M_{\mathcal{A}}$ are applied to messages and agents in τ_{ag1}, respectively.

Theorem 4. *Given three LAIPs τ_{ag1}, τ_{ag2} and τ_{ag3}:*

$$\tau_{ag1} \preceq \tau_{ag2} \wedge \tau_{ag2} \preceq \tau_{ag3} \implies \tau_{ag1} \preceq \tau_{ag3}$$

Proof. By construction from Definition 12. □

Theorem 5. *Given three LAIPs τ_{ag1}, τ_{ag2} and τ_{ag3}:*

$$\tau_{ag1} \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}}\rangle} \tau_{ag2} \wedge \tau_{ag2} \preceq_{\langle M'_{\mathcal{M}}, M'_{\mathcal{A}}\rangle} \tau_{ag3}$$

$$\implies$$

$$\tau_{ag1} \preceq_{\langle M''_{\mathcal{M}}, M''_{\mathcal{A}}\rangle} \tau_{ag3} \wedge M''_{\mathcal{M}} = M'_{\mathcal{M}} \circ M_{\mathcal{M}} \wedge M''_{\mathcal{A}} = M'_{\mathcal{A}} \circ M_{\mathcal{A}}$$

Proof. From Definition 13 we know that

$$\tau_{ag1} \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}}\rangle} \tau_{ag2} \iff < M_{\mathcal{M}}, M_{\mathcal{A}} > (\tau_{ag1}) \preceq \tau_{ag2}$$

From Definition 13 we know that

$$\tau_{ag2} \preceq_{\langle M'_{\mathcal{M}}, M'_{\mathcal{A}}\rangle} \tau_{ag3} \iff < M'_{\mathcal{M}}, M'_{\mathcal{A}} > (\tau_{ag2}) \preceq \tau_{ag3}$$

Thanks to Theorem 4, we can deduce that

$$< M'_{\mathcal{M}} \circ M_{\mathcal{M}}, M'_{\mathcal{A}} \circ M_{\mathcal{A}} > (\tau_{ag1}) \preceq \tau_{ag3}$$

that is equivalent to $\tau_{ag1} \preceq_{\langle M'_{\mathcal{M}} \circ M_{\mathcal{M}}, M'_{\mathcal{A}} \circ M_{\mathcal{A}}\rangle} \tau_{ag3}$. □

An algorithm for conformance. Given the definitions 12 and 13, a first question that may arise is whether there exists an algorithm for deciding if the compliance relation holds for a pair of trace expressions, and, in case of the more general notion of conformance modulo mappings, if such mappings can be computed. Unfortunately, the problem is undecidable even for the simpler conformance relation of Definition 12; this can be derived by the fact that a context-free grammar can be encoded into a trace expression, and that the problem of inclusion between context-free languages (which is known to be undecidable) can be reduced to the conformance problem between two trace expressions. Despite this negative result, it is still interesting to investigate the existence of algorithms which are sound (even though not complete) w.r.t. the definition of conformance between trace expressions.

We define the merging of two maps in the following way: let $M_{\mathcal{M}} : \mathcal{M}1 \rightarrow \mathcal{M}2$ and $M'_{\mathcal{M}} : \mathcal{M}1' \rightarrow \mathcal{M}2'$ be two maps among messages:

if $\exists_{msg \in \mathcal{M}1 \cap \mathcal{M}1'} . M_{\mathcal{M}}(msg) \neq M'_{\mathcal{M}}(msg)$

then $merge(M_{\mathcal{M}}, M'_{\mathcal{M}}) = \emptyset$

else $merge(M_{\mathcal{M}}, M'_{\mathcal{M}}) = M''_{\mathcal{M}} : \mathcal{M}1 \cup \mathcal{M}1' \rightarrow \mathcal{M}2 \cup \mathcal{M}2'$ such that $M''_{\mathcal{M}} = M_{\mathcal{M}} \cup M'_{\mathcal{M}}$.

In other words, merging two maps consists in computing the union of the elements in the maps, unless there is some conflict, namely, some element is mapped to two different elements in the two maps. In this case, the maps cannot be merged (the merged map is empty). For instance, if $M_{\mathcal{M}} = \{msg_1 \mapsto msg_2, msg_3 \mapsto msg_4\}$ and $M'_{\mathcal{M}} = \{msg_3 \mapsto msg_4, msg_5 \mapsto msg_6\}$, the merged map is $M''_{\mathcal{M}} = \{msg_1 \mapsto msg_2, msg_3 \mapsto msg_4, msg_5 \mapsto msg_6\}$. If $M_{\mathcal{M}} = \{msg_1 \mapsto msg_2\}$ and $M'_{\mathcal{M}} = \{msg_1 \mapsto msg_3\}$, their merged map is empty.

The same definition can be adopted for merging maps of agents.

Given two maps $M_{\mathcal{M}}$ and $M_{\mathcal{A}}$, a sending interaction $\overset{msg}{\Longrightarrow}_R$ can substitute a sending interaction $\overset{msg'}{\Longrightarrow}_{R'}$ in the context of $M_{\mathcal{M}}$ and $M_{\mathcal{A}}$ iff $merge(\{msg \mapsto msg'\}, M_{\mathcal{M}}) \neq \emptyset$ and $merge(\{R \mapsto R'\}, M_{\mathcal{A}}) \neq \emptyset$. The definition for a receiving interaction $\overset{msg}{\Longleftarrow}_S$ substituting a receiving interaction $\overset{msg'}{\Longleftarrow}_{S'}$ is similar.

The computation of $\tau_{ag1} \leq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau'_{ag2}$ is carried out by the "isConformant" algorithm. The algorithm starts from two initial agent and message maps, and incrementally adds to them those mappings which are necessary to ensure agents interoperability. Consequently, it is possible to obtain partial maps where some messages and agents have not been mapped to anything at the end of the computation. Partial maps must be completed (namely, they must become total maps and be defined on all the elements in their domain), in order to be used in practice. Completion can be achieved by adding dummy elements in the range, and associate the elements in the domain that had no corresponding element in the range, with such dummy elements. The completion step is necessary to ensure that, when actually used to substitute $ag1 \in mas$ to $ag2 \in mas'$, the maps returned by the algorithm can be applied to all the agents and messages appearing in τ_{ag1}. The isConformant algorithm

operates by cases, and the following implication holds:

$$\tau_{ag1} \leq_{\langle cM_{\mathcal{M}}, cM_{\mathcal{A}} \rangle} \tau'_{ag2} \Longleftarrow$$

$\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle$ = isConformant(τ_{ag1}, τ'_{ag2}, $\emptyset$, $\{ag1 \mapsto ag2\}$) **and** $\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle \neq \langle \emptyset, \emptyset \rangle$ **and** $complete(M_{\mathcal{M}}) = cM_{\mathcal{M}}$ **and**
$complete(M_{\mathcal{A}}) = cM_{\mathcal{A}}$, where *complete* is the map completion step sketched above.
Conformance can be lifted to global protocols:

$$\tau \leq \tau' \iff \forall_{ag_i \in mas} . \exists_{ag'_j \in mas'} . \tau_{ag_i} \leq \tau'_{ag'_j}$$

$$\tau \leq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau' \iff \forall_{ag_i \in mas} . \exists_{ag'_j \in mas'} . \tau_{ag_i} \leq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau'_{ag'_j}$$

Given a projected trace expression τ_{ag},

$$active_int(\tau_{ag}) = \{ \overset{msg}{\Longrightarrow}_R \mid \exists \tau'_{ag}, \tau_{ag} \overset{\overset{msg}{\Longrightarrow}_R}{\rightarrow} \tau'_{ag} \}.$$

Given a projected trace expression τ_{ag},

$$passive_int(\tau_{ag}) = \{ \overset{msg}{\Longleftarrow}_S \mid \exists \tau'_{ag}, \tau_{ag} \overset{\overset{msg}{\Longleftarrow}_S}{\rightarrow} \tau'_{ag} \}.$$

In Chapter 12 we will present the concept of over-approximation of a trace expression in mode detail. For now, it is enough to know that the over-approximation of a trace expression is a trace expression recognizing a superset of the traces recognized by the first one. As we will see, we use over-approximation to remove expansive terms from inside trace expressions. We need this additional step because our approach does not support the comparison between two expansive terms.

Theorem 6. *Given an expansive concatenation $\tau_{ag} = \tau_1 \cdot \tau_2$ and its over-approximation $\widetilde{\tau_{ag}} = \widetilde{\tau_1 \cdot \tau_2}$ (obtained using the algorithm presented in Chapter 12):*

1. $passive_int(\tau_2) = \{\} \implies \tau_1 \cdot \tau_2 \leq_{\langle Id_{\mathcal{M}}, Id_{\mathcal{A}} \rangle} \widetilde{\tau_1 \cdot \tau_2}$
2. $active_int(\tau_2) = \{\} \implies \widetilde{\tau_1 \cdot \tau_2} \leq_{\langle Id_{\mathcal{M}}, Id_{\mathcal{A}} \rangle} \tau_1 \cdot \tau_2$

where $Id_{\mathcal{M}}$ and $Id_{\mathcal{A}}$ are the identity functions which map each message and each agent in itself, respectively.

Proof. Since an expansive concatenation is used to balance sequences of interactions, e.g. $\{msg_1^n msg_2^n\}$, the over-approximation obtained using the algorithm defined in Chapter 12 is the regular superset $\{msg_1^* msg_2^*\}$. The idea is to identify which agent decides the number of interactions for the queue τ_2. An agent sending n messages is interoperable in a context where the receiver is supposed to receive any number $*$. On the other hand, an agent which is able to receive any number $*$ of messages is interoperable in a context where the sender will send exactly n messages. When $passive_int(\tau_2) = \{\}$, we fall in the first scenario, where ag is able to send n messages in the queue τ_2, and the receiver in $\widetilde{\tau_1 \cdot \tau_2}$ is able to receive any number $*$ of that messages. Thus ag can interoperate in the over-approximated scenario. In the same way, when $active_int(\tau_2) = \{\}$, we fall in the second scenario, where ag is able to receive

any number $*$ of messages in the over-approximated scenario, and the sender in $\tau_1 \cdot \tau_2$ is able to send a fixed number n of that messages. Thus ag can interoperate in the expansive scenario. Since the over-approximation preserves the structure of the sub-terms and the sets of agents and interactions, we know that among the possible maps that can be used there are also the identity functions $Id_{\mathcal{M}}, Id_{\mathcal{A}}$ (we are only changing the number of messages exchanged, we do not change the structure of our protocols during the over-approximation). □

Conjecture 11.3.1 (Soundness)**.** *The algorithmic implementation of the conformance test is sound. If given two LAIPs τ_{ag} and $\tau'_{ag'}$, the algorithm returns a couple of maps $M_{\mathcal{M}}$ and $M_{\mathcal{A}}$, we know that $\tau_{ag} \leq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau'_{ag'}$ is satisfied and consequently we can substitute the agent ag' using ag preserving the system interoperability.*

Claim 11.3.1 (No Completeness)**.** *The algorithmic implementation of the conformance test is not complete.*

Proof. When we implement the conformance test for the expansive trace expressions we tackle only a subset of the possible combinations. For instance, we are not able to compare an expansive concatenation with an expansive shuffle, consequently, it could be possible to show how two expansive trace expressions representing the same LAIP using different operators are considered not conformant by our algorithm (even if they are the same). □

11.4 Conformance algorithm: pseudo-code

The notion of $\tau_{ag1} \leq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau'_{ag2}$ is defined by cases.

NON EXPANSIVE TRACE EXPRESSIONS The pseudo-code for the conformance check (constructed on top of Definition 13) is described by the following recursive function:
function isConformant
input: two non-expansive trace expressions τ_{ag1} and τ'_{ag2}, a (partial) map among messages $M_{\mathcal{M}}$, a (partial) map among agents $M_{\mathcal{A}}$
output: one among the many possible couples of maps $\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle$ that allow $ag1$ to substitute $ag2$ ($\langle \emptyset, \emptyset \rangle$ means that the substitution is not possible)
if $\tau_{ag1} = \epsilon$ and $\tau'_{ag2} = \epsilon$, or a cycle has been detected
then return $\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle$
else if the three conditions below hold

1. each sending interaction $\overset{msg_A}{\Longrightarrow}_{agA}$ s.t. $\tau_{ag1} \overset{\overset{msg_A}{\Longrightarrow}_{agA}}{\longrightarrow} \tau_{1_{ag1}}$ can substitute one sending interaction $\overset{msg_B}{\Longrightarrow}_{agB}$ s.t. $\tau'_{ag2} \overset{\overset{msg_B}{\Longrightarrow}_{agB}}{\longrightarrow} \tau'_{1_{ag2}}$ in the context of $M_{\mathcal{M}}$ and $M_{\mathcal{A}}$, $M'_{\mathcal{M}}$ and $M'_{\mathcal{A}}$ are the non-empty maps obtained by merging the mappings generated by such substitutions with $M_{\mathcal{M}}$ and $M_{\mathcal{A}}$ respectively, and $\langle SM_{\mathcal{M}}, SM_{\mathcal{A}} \rangle$ = isConformant($\tau_{1_{ag1}}$, $\tau'_{1_{ag2}}$, $M'_{\mathcal{M}}$, $M'_{\mathcal{A}}$) $\neq \langle \emptyset, \emptyset \rangle$;

2. each receiving interaction $\overset{msg_B}{\Longleftarrow}_{agB}$ s.t. $\tau'_{ag2} \overset{\overset{msg_B}{\Longleftarrow}_{agB}}{\longrightarrow} \tau'_{1_{ag2}}$ can be substituted by one receiving interaction $\overset{msg_A}{\Longleftarrow}_{agA}$ s.t. $\tau_{ag1} \overset{\overset{msg_A}{\Longleftarrow}_{agA}}{\longrightarrow} \tau_{1_{ag1}}$, $M'_{\mathcal{M}}$ and $M'_{\mathcal{A}}$ are the non-empty maps obtained by merging the mappings generated by such substitutions with $M_{\mathcal{M}}$ and $M_{\mathcal{A}}$ respectively, and $\langle RM_{\mathcal{M}}, RM_{\mathcal{A}}\rangle$ = isConformant($\tau_{1_{ag1}}$, $\tau'_{1_{ag2}}$, $M'_{\mathcal{M}}$, $M'_{\mathcal{A}}$) $\neq \langle\emptyset, \emptyset\rangle$;
3. if there is one sending interaction available for τ'_{ag2}, there must be also at least one sending interaction available in τ_{ag1} that can be substituted by it in the context of $M_{\mathcal{M}}$ and $M_{\mathcal{A}}$.

then return $\langle merge(SM_{\mathcal{M}}, RM_{\mathcal{M}}), merge(SM_{\mathcal{A}}, RM_{\mathcal{A}})\rangle$
else return $\langle\emptyset, \emptyset\rangle$
endif

We can now state that

$$\tau_{ag1} \leq_{\langle cM_{\mathcal{M}}, cM_{\mathcal{A}}\rangle} \tau'_{ag2} \Longleftarrow$$

$\langle M_{\mathcal{M}}, M_{\mathcal{A}}\rangle$ = isConformant(τ_{ag1}, τ'_{ag2}, $\emptyset$, $\{ag1 \mapsto ag2\}$) **and** $\langle M_{\mathcal{M}}, M_{\mathcal{A}}\rangle \neq \langle\emptyset, \emptyset\rangle$ **and** $complete(M_{\mathcal{M}}) = cM_{\mathcal{M}}$ **and**
$complete(M_{\mathcal{A}}) = cM_{\mathcal{A}}$

We point out that the couple of maps $\langle cM_{\mathcal{M}}, cM_{\mathcal{A}}\rangle$ that make τ_{ag1} and τ'_{ag2} conformant is just one of the possibly many existing ones. In fact, in the pseudo-code above, there are some non-deterministic choices due to the choice of one sending interaction in (1) and one receiving interaction in (2), among possibly many available ones. In order to obtain all the possible maps, we should "backtrack" on all the choice points and collect the solutions we obtain making different choices. This "all-solutions" approach is actually supported by the implemented algorithm, which – being implemented in Prolog – makes the exploitation of backtracking extremely easy to program.

The conformance check described in the pseudo-code is inspired by the one described in (Baldoni et al., 2005b, 2006), but it tackles two problems that did not arise there. One is related with mappings $M_{\mathcal{M}}$ and $M_{\mathcal{A}}$ which are incrementally extended while the conformance check goes on, and are finally output when the algorithm terminates. The second is that in (Baldoni et al., 2005b, 2006) protocols are not recursive and hence taking care of cycles is not necessary. Since we may handle recursive trace expressions, we have to remember the history of the subtraces analyzed so far in order to identify if a subtrace has already been checked. Like the trace expressions operational semantics, also the conformance check is defined in a coinductive way and, when it discovers that two trace expressions have been already checked previously, it can terminate with a positive answer and avoid the infinite loop.

When at least one of the trace expressions involved in the conformance check is expansive, the algorithm becomes much more complex: conformance checking of expansive trace expressions may be undecidable, and we may need to over-approximate the expansive trace expression τ' with a non-expansive

one, to check whether τ is conformant to it (Theorem 6).

For what concerns expansive trace expressions, unfortunately we cannot be precise in the same way as we are for the non expansive counterpart.

In Theorem 6, given an expansive concatenation $\tau_1 \cdot \tau_2$, we showed that

1. $passive_int(\tau_2) = \{\} \implies \tau_1 \cdot \tau_2 \preceq_{\langle Id_{\mathcal{M}}, Id_{\mathcal{A}} \rangle} \widetilde{\tau_1 \cdot \tau_2}$
2. $active_int(\tau_2) = \{\} \implies \widetilde{\tau_1 \cdot \tau_2} \preceq_{\langle Id_{\mathcal{M}}, Id_{\mathcal{A}} \rangle} \tau_1 \cdot \tau_2$

Following Theorem 6, we can compare two expansive trace expressions through their over-approximation.

function isConformant
input: one non-expansive trace expression τ_{ag1} and one expansive concatenation $\tau'_{ag2} = \tau'_1 \cdot \tau'_2$, a (partial) map among messages $M_{\mathcal{M}}$, a (partial) map among agents $M_{\mathcal{A}}$
output: one among the many possible couples of maps $\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle$ that allow $ag1$ to substitute $ag2$ ($\langle \emptyset, \emptyset \rangle$ means that the substitution is not possible)
if $active_int(\tau'_2) = \{\}$
then return isConformant(τ_{ag1}, $\widetilde{\tau'_{ag2}}$, $M_{\mathcal{M}}$, $M_{\mathcal{A}}$)
else return $\langle \emptyset, \emptyset \rangle$

The pseudo-code above is the implementation of the following intuition. Given an expansive concatenation $\tau_1 \cdot \tau_2$, we have that

$$active_int(\tau_2) = \{\} \implies \widetilde{\tau_1 \cdot \tau_2} \preceq_{\langle Id_{\mathcal{M}}, Id_{\mathcal{A}} \rangle} \tau_1 \cdot \tau_2 \qquad \text{(Theorem 6)}$$

And, given three trace expressions τ_{ag1}, τ_{ag2} and τ_{ag3} we have that

$$\tau_{ag1} \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau_{ag2} \wedge \tau_{ag2} \preceq_{\langle M'_{\mathcal{M}}, M'_{\mathcal{A}} \rangle} \tau_{ag3} \implies$$
$$\tau_{ag1} \preceq_{\langle M''_{\mathcal{M}}, M''_{\mathcal{A}} \rangle} \tau_{ag3} \wedge M''_{\mathcal{M}} = M'_{\mathcal{M}} \circ M_{\mathcal{M}} \wedge M''_{\mathcal{A}} = M'_{\mathcal{A}} \circ M_{\mathcal{A}} \qquad \text{(Theorem 5)}$$

Consequently, we can conclude that, given an expansive concatenation $\tau_1 \cdot \tau_2$ and a non-expansive trace expression τ, we have that

$$active_int(\tau_2) = \{\} \wedge \tau \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \widetilde{\tau_1 \cdot \tau_2}$$
$$\implies$$
$$\tau \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau_1 \cdot \tau_2$$

function isConformant
input: one expansive concatenation $\tau_{ag1} = \tau_1 \cdot \tau_2$ and one non-expansive trace expression τ'_{ag2} , a (partial) map among messages $M_{\mathcal{M}}$, a (partial) map among agents $M_{\mathcal{A}}$
output: one among the many possible couples of maps $\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle$ that allow $ag1$ to substitute $ag2$ ($\langle \emptyset, \emptyset \rangle$ means that the substitution is not possible)
if $passive_int(\tau_2) = \{\}$
then return isConformant($\widetilde{\tau_{ag1}}$, τ'_{ag2}, $M_{\mathcal{M}}$, $M_{\mathcal{A}}$)
else return $\langle \emptyset, \emptyset \rangle$

The pseudo-code above is the implementation of the following intuition. Given an expansive concatenation $\tau_1 \cdot \tau_2$, we have that

$$passive_int(\tau_2) = \{\} \implies \tau_1 \cdot \tau_2 \preceq_{\langle Id_{\mathcal{M}}, Id_{\mathcal{A}} \rangle} \widetilde{\tau_1 \cdot \tau_2} \qquad \text{(Theorem 6)}$$

And, given three trace expressions τ_{ag1}, τ_{ag2} and τ_{ag3} we have that

$$\tau_{ag1} \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau_{ag2} \wedge \tau_{ag2} \preceq_{\langle M'_{\mathcal{M}}, M'_{\mathcal{A}} \rangle} \tau_{ag3} \implies$$

$$\tau_{ag1} \preceq_{\langle M''_{\mathcal{M}}, M''_{\mathcal{A}} \rangle} \tau_{ag3} \wedge M''_{\mathcal{M}} = M'_{\mathcal{M}} \circ M_{\mathcal{M}} \wedge M''_{\mathcal{A}} = M'_{\mathcal{A}} \circ M_{\mathcal{A}} \quad \text{(Theorem 5)}$$

Consequently, we can conclude that, given an expansive concatenation $\tau_1 \cdot \tau_2$ and a non-expansive trace expression τ, we have that

$$passive_int(\tau_2) = \{\} \wedge \widetilde{\tau_1 \cdot \tau_2} \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau$$

$$\implies$$

$$\tau_1 \cdot \tau_2 \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau$$

The conformance with an expansive shuffle is very similar. It is enough to show – in the same way as for expansive concatenations – that given an expansive $\tau_1 | \tau_2$, we have that $\tau_1 | \tau_2 \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \widetilde{\tau_1 | \tau_2}$.

11.5 Implementation and Experiments

Given two MAS *mas* and *mas′* ruled by $\tau = GAIP(mas)$ and $\tau' = GAIP(mas')$ respectively, the steps for using an agent involved in *mas* inside *mas′* are the following:

1. identify the agent $ag1 \in mas$ to be used in mas';
2. generate the set of agents and maps that ensure τ_{ag1} conformance $\{(ag2, \langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle) | ag2 \in mas', \tau_{ag1} \preceq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau'_{ag2}\}$;
3. select one pair of agents and maps $(ag2, \langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle)$ from the set, based on domain-dependent criteria that might involve message similarity, similar behaviours of the mapped agents, and so on;
4. generate an interface i for $ag1$ driven by $(ag2, \langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle)$;
5. substitute $ag2$ in mas' with the "interfaced version" of $ag1$: the agents in the MAS obtained via this substitution still respect τ', because of step (2).

The notion of *interface* introduced in step (4) is related to the actual use of the maps generated during the conformance check and it is meant as a logical component offering a "bridging service", that can be implemented in many different ways. Hence, the interface i generated in step (4) realizes the map-driven translations needed by agents in *mas* and *mas′* to interoperate. Each time $ag1 \in mas$ performs the action of sending a message msg_1 to an agent $ag1_r \in mas$, the interface i "intercepts" (from a logical point of view) msg_1, translates it into $M_{\mathcal{M}}(msg_1) = msg_2$, and forwards msg_2 to $M_{\mathcal{A}}(ag1_r) \in mas'$. In the same way, each time an agent $ag2_s \in mas'$ sends a message msg_3 to $ag2 \in mas'$, the interface intercepts msg_3, looks for a message msg_4 that $ag1$ can receive in the current protocol state s.t. $M_{\mathcal{M}}(msg_4) = msg_3$, translates msg_3 into msg_4, and forwards it to $ag1$.

As an example, let us consider a GAIP τ representing the protocol where an agent *Buyer* (*Buy*) asks to an agent *Seller* (*Sel*) for a resource and if the resource is available, the *Seller* can give it in exchange of money, otherwise the *Seller* informs the *Buyer* of the unavailability.

$$\tau = (Buy \overset{res?}{\Longrightarrow} Sel):$$
$$(Sel \overset{res}{\Longrightarrow} Buy{:}Buy \overset{money}{\Longrightarrow} Sel{:}\epsilon \vee Sel \overset{no}{\Longrightarrow} Buy{:}\tau)$$

Let us consider another similar GAIP τ' defining a book-shop protocol where an agent *Client* (*Cl*) asks for a book to an agent *BookShop*, and again if the book is available the *BookShop* (*Shop*) agent sells it for a given amount of euros, otherwise a *no_avbl* message is returned.

$$\tau' = (Cl \overset{book?}{\Longrightarrow} Shop):$$
$$((Shop \overset{book}{\Longrightarrow} Cl{:}Cl \overset{euros}{\Longrightarrow} Shop{:}\epsilon) \vee (Shop \overset{no_avbl}{\Longrightarrow} Cl{:}\tau'))$$

We want to check if τ is conformant to τ', $\tau \leq \tau'$. From the definition of conformance between global protocols, for each agent $ag \in \tau$ we must find at least one agent $ag' \in \tau'$ s.t. $\tau_{ag} \leq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau'_{ag'}$. First of all, we generate the local perspectives LAIPs of τ and τ' through projection.

$$\tau_{Buy} = (\overset{res?}{\Longrightarrow}_{Sel}){:}((\overset{res}{\Longleftarrow}_{Sel} : \overset{money}{\Longrightarrow}_{Sel} :\epsilon) \vee (\overset{no}{\Longrightarrow}_{Sel} :\tau_{Buy}))$$
$$\tau_{Sel} = (\overset{res?}{\Longleftarrow}_{Buy}){:}((\overset{res}{\Longrightarrow}_{Buy} : \overset{money}{\Longleftarrow}_{Buy} :\epsilon) \vee (\overset{no}{\Longleftarrow}_{Buy} :\tau_{Sel}))$$
$$\tau'_{Cl} = (\overset{book?}{\Longrightarrow}_{Shop}){:}((\overset{book}{\Longleftarrow}_{Shop} : \overset{euros}{\Longrightarrow}_{Shop} :\epsilon) \vee (\overset{no_avbl}{\Longleftarrow}_{Shop} :\tau'_{Cl}))$$
$$\tau'_{Shop} = (\overset{book?}{\Longleftarrow}_{Cl}){:}((\overset{book}{\Longrightarrow}_{Cl} : \overset{euros}{\Longleftarrow}_{Cl} :\epsilon) \vee (\overset{no_avbl}{\Longrightarrow}_{Cl} :\tau'_{Shop}))$$

Then we apply the rules deriving from Definition 13, obtaining that (Figure 11.1c) $\tau_{Buy} \leq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau'_{Cl}$ with $M_{\mathcal{M}} = \{res? \mapsto book?, res \mapsto book, money \mapsto euros, no \mapsto no_avbl\}$ and $M_{\mathcal{A}} = \{Buy \mapsto Cl, Sel \mapsto Shop\}$ and (Figure 11.1d) $\tau_{Sel} \leq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau'_{Shop}$ with the same maps. From this, we derive that $\tau \leq_{\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle} \tau'$.

In this example we had no prior knowledge on possibly correct mappings. In many real scenarios, however, some of the correct mappings among messages and agents, respectively, are known in advance. The values $\emptyset$, $\{ag1 \mapsto ag2\}$ used to initialize the maps in the definition of $\tau_{ag1} \leq_{\langle cM_{\mathcal{M}}, cM_{\mathcal{A}} \rangle} \tau'_{ag2}$ **iff** $\langle M_{\mathcal{M}}, M_{\mathcal{A}} \rangle$ = isConformant(τ_{ag1}, τ'_{ag2}, $\emptyset$, $\{ag1 \mapsto ag2\}$) correspond to the worst case where the developer has no knowledge at all about the possible correct mappings, and wants to generate all of them for further inspection. If the developer knows that the initial associations modelled by $M_{\mathcal{M}_0}$ and $M_{\mathcal{A}_0}$ must hold, he/she can run isConformant(τ_{ag1}, τ'_{ag2}, $M_{\mathcal{M}_0}$, $M_{\mathcal{A}_0}$), forcing the algorithm to extend such initial knowledge with new associations or to answer that, with these maps, the protocols are not conformant. This would be the case for example in Software Product Line applications and in evolving IoT scenarios where the developer knows which components in the previous product version/system should be replaced by which in the new one, and wants to check if it is possible, respecting the existing communication protocols.

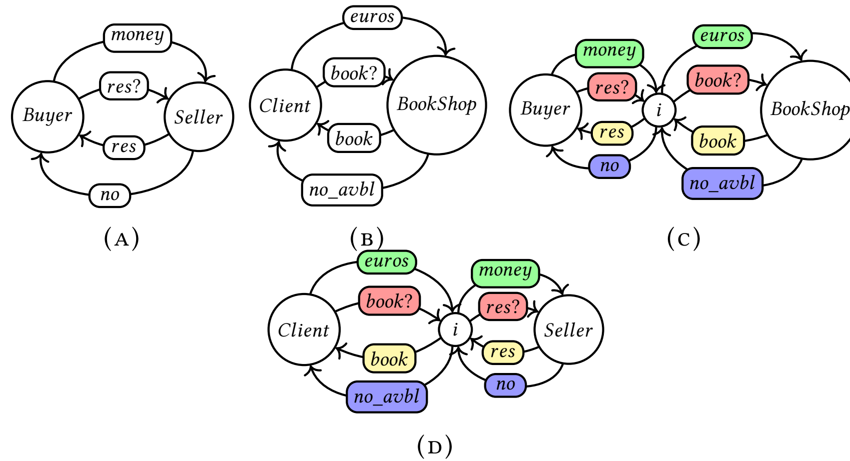

FIGURE 11.1. (a) and (b) MAS presented in the example; (c) *Buyer* substitutes *Client* through the interface i driven by $M_{\mathcal{A}} = \{Buyer \mapsto Client, Seller \mapsto BookShop\}$, $M_{\mathcal{M}} = \{res? \mapsto book?, res \mapsto book, money \mapsto ack, no \mapsto no_avbl\}$); (d) *Seller* substitutes *BookShop* with an interface driven by the same maps.

Although introducing the notion of interface for showing how we can use the mappings generated during the conformance check (step (5) presented in Section 11.1 deals with exploiting mappings in practice) makes the presentation almost simple and intuitive, the actual implementation of such an interface requires to take care of many aspects dependent on the adopted MAS framework. Step (5) is in fact heavily application-dependent and can be faced in different ways, depending on the applications constraints: it would be possible for example to insert a “translator agent” into *mas′*, that intercepts and manages all interactions involving *ag*; or to create a wrapper for agent *ag* that allows it to automatically translate incoming and outgoing messages according to $M_{\mathcal{M}}$; or even to automatically modify *ag*’s source code, if available, to hard-wire the $M_{\mathcal{M}}$ mapping at source code level. Whatever the approach is, all the agents in *mas′* should be aware that *ag′* has been substituted by *ag*, in order to handle communication properly.

To demonstrate how to exploit the maps generated by the conformance testing to substitute JADE agents with other JADE agents, we adopted an automatic source code translation approach. The methodology we followed consists in three steps:

1st STEP, CONFORMANCE CHECKING (CORRESPONDING TO STEPS (1),(2) AND (3) PRESENTED AT THE VERY BEGINNING OF THIS SECTION) The algorithm presented in Section 11.3 is fully implemented in SWI-Prolog. Prolog has been chosen thanks to its built-in support to cyclic terms, coinduction, and for the possibility to use backtracking for generating all the existing maps. The implementation of the algorithm is $<$ 400 LOC.

2nd STEP, SUBSTITUTION (CORRESPONDING TO STEP (4)) Let us suppose that $ag1 \in mas$ can substitute $ag2 \in mas'$ with maps $M_{\mathcal{A}}$ and $M_{\mathcal{M}}$. For demonstrating how substitution can be put into practice we have opted for a basic approach where we apply the maps to the source code of $ag1$, and we use the modified source code $map(ag1)$ instead of the source code of $ag2$ in mas': we operate on the Java file containing the JADE class implementing $ag1$ and we substitute all the occurrences of ag_i with $M_{\mathcal{A}}(ag_i)$ and all occurrences of msg_i with $M_{\mathcal{M}}(msg_i)$. This substitution step is also implemented in SWI-Prolog (< 30 LOC).

3rd STEP, EXECUTION (CORRESPONDING TO STEP (5)) We recompile $map(ag1)$ and we execute mas' with $map(ag1)$ instead of $ag2$. Despite being simple and applicable only when the source code is available, this approach demonstrates how we can actually use the maps generated during the conformance check, and has been adopted for all the examples shown in this chapter.

11.6 Discussion

In this chapter we have presented a conformance modulo mapping algorithm suitable for checking conformance between local protocols specified as (projected) trace expressions, together with its implementation and usage example. The chapter presents a general solution to the problem with as few constraints as possible, to make it reusable in as many situations as possible, but the actual scenarios where we believe that our approach can be more profitably exploited involve conformance between different versions of the same LAIP or LAIPs which are known to be similar, like the ones presented in Sections 11.1 and 11.5. As another example, a self-driving car may interact with other cars, lights, etc., according to the current road norms (LAIP τ_1). Norms change and the new LAIP to which the car must conform, becomes τ_2. Which transformations (mappings) should we implement over τ_1 to ensure it is syntactically conformant to τ_2? The developer in charge of migrating τ_1 to τ_2 can use our algorithm for having guarantees on the syntactic compliance, although he/she cannot have guarantees that semantics is preserved: a human is required to finally select and validate the produced mappings. We think that semantic compliance will never be fully automatized, and for this reason we expect that our algorithm should be used in scenarios where LAIPs should not be re-aligned frequently.

Part V

Combining static and runtime verification

Runtime verification and Static verification are two different verification approaches. Nonetheless, it is not surprising that during the last decade many works have been studying the possible combination of these two disciplines. Also during the Ph.D. program, we looked into these aspects and we propose two possible approaches.

In Chapter 12, we present our first attempt, where we show how to translate a trace expression into an equivalent Büchi Automaton in order to verify statically our specifications using the SPIN model checker.

In Chapter 13, we present the work that has been done during the visit to the University of Liverpool. The main idea consists in using trace expressions to represent the abstract environment used for model checking the MAS. In this way, after the model checking process has finished, we can still use the trace expression to verify at runtime the MAS – with the real environment – in order to understand if there are violations of the assumptions that have been done during the static verification process.

12 *Trace expressions model checking*

"Trust, but Verify."
- Old Russian proverb

In this chapter, we propose an algorithm to check LTL properties satisfiability on trace expressions. To do this, we show how to translate a trace expression into a Büchi Automaton in order to realize an Automata-Based Model Checking. *We show that this translation generates an* over-approximation *of the trace expression leading us to obtain a sound procedure to verify LTL properties. Once we have statically checked a set of LTL properties, we can conclude that: (1) the trace expression is formally correct (2) since we use this trace expression to generate monitors checking the runtime behavior of the system, the LTL properties verified by this trace expression are also verified by the monitored system.*

The contents of this chapter are published in (Ferrando, 2019)

12.1 Introduction

As already introduced in Section 3.5.2, Runtime Verification (RV) is a software verification technique that complements formal static verification (like Model Checking (Clarke, Grumberg, and Peled, 2001; Holzmann, 2002; Merz, 2000; Visser et al., 2003)) and testing (Broy et al., 2005b; Chow, 1978; Myers, Sandler, and Badgett, 2011).

When the system we want to verify becomes larger, model checking it (as well as the environment where it is immersed) becomes quickly intractable. In these scenarios, a valid alternative is RV. The main difference with respect to standard static verification is the stage when it is applied, which is at execution time. In fact, in RV we do not need to simulate all possible paths that the system may generate during its execution, but we limit the analysis directly to the paths exposed and generated by the system during its real execution. As a consequence, RV could be more suitable and applicable than static verification in black-box scenarios, where there is no access to the source code of the system we want to verify.

Since we generate a monitor starting from a formal specification (Section 4.4), we represent statically what we will check dynamically. If the static representation of the allowed event traces were error-free, we would be sure that monitoring a system according to that representation, would allow us to intercept all and only the possible violations due to unexpected or unwanted sequences of events. Unfortunately, our static representation might contain design and formalization errors too, making unproductive to monitor the system behavior. If we were able to verify properties of the static representation before using it for the dynamic monitoring, the runtime verification would lead to more controlled and meaningful results. In the case of trace expressions, a possible way for obtaining this static check before the use at runtime is verifying whether all the traces satisfy a LTL property (Pnueli, 1977). For instance, a common useful property to be checked could be $a \Rightarrow \Diamond b$ which says that if a takes place *sooner or later*, b will take place as well. Considering that the monitors used to verify the system are generated starting from a trace expression[1], we can conclude that all properties satisfied by trace expressions are either satisfied by the monitored system, or their violation is recognized by the monitor. This represents an important aspect because we can fuse static and dynamic approaches obtaining the best of both: the formal verification at the *static* level and the runtime monitoring at the *dynamic* level.

For instance, the combination of static and runtime verification can simplify – reduce the size of – our monitors. If we were able to check statically a part of our specification, we could verify dynamically at runtime only what we are not able to check at static time (Hinrichs, Sistla, and Zuck, 2014). Or also, if we want to reduce the state space analyzed by a static verifier, we can make assumptions and relax the model that is being validated. But, in this way, we can not be

[1]To be more precise, in our implementation the monitors interpret the trace expression by implementing the transition rules which define the trace expression semantics.

sure the real system is compliant with our assumptions, and if it is not, it would make the static verification so obtained useless[2]. Combining the use of a monitor to the static verifier, we can simplify the model verified statically, and we can add a monitor at runtime in order to recognize assumption violations with respect to our model (Chapter 13). These are just possible advantages in combining static and runtime verification. In this work, we focus on another possible combination of static and runtime verification, showing how we can **first verify** statically our monitors, and **then**, how to use them to **monitor** our systems. Our research question can be summarized in:

How can we trust our specification before using it to monitor our system?

One possible way of achieving the static verification of our specification (thus, our monitor) is translating it into a model more suitable for static verification purposes. In this work, we will show how to translate our specifications into Büchi Automatons, that are the standard automata version of LTL formulas used in model checking. We chose Büchi Automatons because they are a standard representation supported by most of the existing model checker (Vardi, 2007). In particular, we will use the SPIN model checker (Holzmann, 1991, 1997, 2004) to verify LTL formulas directly on our specifications.

Naturally, if we used LTL for generating our monitor, we would not need to verify anything statically, because the monitor would already denote the properties we want to check. But, LTL could be too limiting when used for RV purposes, and we might need more expressive formalisms that allow defining more complex monitors. As we have already presented in Section 4.5.1, the formalism we chose to use for defining our monitors is more expressive than LTL[3]. Thus, if we want to be sure that our "complex" monitor still satisfies a set of LTL properties, we need a way to verify them on it. Finally, we can conclude that, even though we have a complex specification that is used to verify at runtime complex properties on our system, we are still able to guarantee that this specification satisfies a given set of LTL properties (without being limited to only those).

12.2 State of the art

In (Bodden, 2005), Bodden *et al.* define a formalism which allows to build expressive formulae over temporal traces in an intuitive way as well as a complete implementation of that formalism, which instruments any given Java application in bytecode form with appropriate runtime checks. That approach is similar to ours, in fact, both pass through the automata theory to generate monitors for finite prefixes. In (Bodden, 2005), the LTL over *pointcuts* version of the formalism is transformed into monitors (Deterministic Büchi Automata) that are rewritten as advice in in AspectJ[4], as in Chapter 4 where the

[2]Our assumptions might be too strong and we need to relax them (see Chapter 13).

[3]Or better, it is more expressive than LTL when the latter is used for RV purposes, namely LTL_3.

[4]`https://eclipse.org/aspectj/`

trace expression formalism is transformed into the corresponding automata representing the LTL_3[5] monitor.

Session types are used to verify object-oriented languages in (Gay et al., 2010), where the authors extend their work on session types for distributed object-oriented languages in three ways:

1. they attach a session type to a class definition, to specify the possible sequences of method calls;

2. they allow a session type (protocol) implementation to be modularized, *i.e.* partitioned into separately- callable methods;

3. they treat session-typed communication channels as objects, integrating their session types with the session types of classes.

Several papers by Dezani-Ciancaglini, Yoshida *et al.* (Capecchi et al., 2009; Dezani-Ciancaglini et al., 2005; Dezani-Ciancaglini et al., 2006; Dezani-Ciancaglini et al., 2007; Hu, Yoshida, and Honda, 2008) have combined session types, as specifications of protocols on communication channels, with the object-oriented paradigm. A characteristic of all of these works is that a channel is always created and used within a single method call.

COMBINING STATIC AND RUNTIME VERIFICATION In (Ahrendt, Pace, and Schneider, 2016; Chimento et al., 2015), Schneider *et al.* present a tool StaRVOOrs which combines static and runtime verification of Java programs using partial results extracted from static verification to optimize the runtime monitoring process. StaRVOOrs combines the deductive theorem prover KeY (Ahrendt et al., 2016) and the runtime verification tool LARVA (Colombo, Pace, and Schneider, 2009), and uses properties written using the ppDATE specification language which combines the control-flow property language DATE used in the runtime verification tool LARVA with Hoare triples assigned to states. In (Gui et al., 2013), Lin Gui *et al.* propose to combine testing (in particular, hypothesis testing) and model checking (in particular, probabilistic model checking) for non-deterministic systems. Their idea is to apply hypothesis testing to system components which are deterministic and use probabilistic model checking to lift the results through non-determinism. In (Artho and Biere, 2005), Artho *et al.* show how to retain information from static analysis for RV, or to compare the results of both techniques inside the NJuke (Artho et al., 2004) framework for static and dynamic analysis of Java programs. In this framework, a static analyzer looks for faults. Reports are then analyzed by a human, who writes test cases for each kind of fault reported. RV will then analyze the program possibly confirming the fault as a failure or counterexample.

[5] LTL_3 is a a three-valued semantics (Bauer, Leucker, and Schallhart, 2009) for LTL formulas, devised to adapt the standard semantics to RV, to correctly consider the limitation that at runtime only finite traces can be checked.

12.3 Motivations

As anticipated in Section 12.1, RV can be seen as a middle approach between model checking and testing.

More in detail, the main differences between RV and model checking can be summarized as follows (Leucker and Schallhart, 2009):

- *time*, model checking is applied statically on the system and not during its execution;
- *exhaustiveness*, model checking generates all possible executions of the system while RV checks only the executions generated by the system at runtime (language inclusion problem vs word problem);
- *invasiveness*, model checking (generally) is applied to white-box scenarios, when the source code of the system is available, while RV can be used both in white-box and black-box scenarios, when the source code could be unavailable;
- *traces length*, model checking can consider arbitrary positions of a infinite trace while RV considers finite executions of increasing size.

In Section 12.1, we presented RV as a counterpart of testing. In particular, RV is extremely close to a specific form of testing, which is sometimes termed as oracle-based testing. In (Leucker and Schallhart, 2009) the authors compare RV with oracle-based testing. In the following, we propose a revised version of the main differences between RV and oracle-based testing (Leucker and Schallhart, 2009):

- *time*, testing is applied during the development of the system while RV is applied before and after the deployment[6];
- *granularity*, testing can test only the output of the system while RV can go into the details of the system behavior checking not only the observable outputs but also how these outputs have been generated (for instance, RV can tackle the nondeterminism, so even if the observable events are correct, they could have been generated in a nondeterministic way).
- *formalization*, in testing, an oracle is typically defined directly, rather than generated from some high-level specification as happen in RV.

After having compared RV with model checking and test, we are ready to answer at the question: *When RV should be used?* (Leucker and Schallhart, 2009):

- when the complexity of the system makes it intractable for an exhaustive analysis;

[6]RV applied before the deployment is used to check if the system respects our model. RV applied after the deployment is used to monitor the system behavior in order to prevent malfunctions (or at least signalling the user in time to reduce the damage).

- when some information is available only at runtime;
- when a precise description of the environment does not exist;
- when there are security issues in the case of safety-critical systems, where it is useful to monitor properties that have been statically proved or tested, mainly to have a double check.

Now that the main differences among RV, model checking and testing have been presented, we can focus on the motivations of the work.

Supposing we have a system to check, in order to do the RV of it we have to start writing the specification we want to verify. In the next section we will present a possible formalism that can be used to obtain this, but for now, we focus only on the problem. Once we have defined our specification, we can use it to generate a monitor to check the system behavior. If the system does something inconsistent with our specification, the monitor will notice it and will act accordingly (implementation dependant). But, what happen if our specification does not represent our intentions? And, our specification allows us to assert something more about the system? If we were able to check our specification statically, we could resolve the first problem. Consequently, we could also say something more about the system verified.

Since RV does not need to exhaustively check the system behavior, we can define more complex properties to verify (intractable properties in static verification). This greater complexity can lead to mistakes also in the development of our specification compromising the entire RV process. One possible way to solve this problem is by checking directly the specification statically before using it. With such preprocessing we achieve two main advantages:

- *trust*, the RV process is more reliable since we are sure that our complex property satisfies a set of constraints (verifiable statically);
- *propagation*, as long as the system is consistent with the monitor generated by the specification, it is also consistent with the constraints checked statically (by construction).

12.4 Model Checking Trace expressions

In this section, we present the model checking process used to check if a LTL property is satisfied by a trace expression.

Given a trace expression τ, we want to verify if a LTL property φ is satisfied by all the traces recognized by τ. To achieve this, we can follow this 3-steps algorithm:

1. Rewrite τ obtaining its abstraction τ' (*over-approximation*). Following the definition of $[\![\tau]\!]$ (Definition 4.2.3), given as the set of traces denoted by the trace expression τ, we have that $[\![\tau']\!] \supseteq [\![\tau]\!]$.
2. Translate τ' in the equivalent Büchi Automaton $B_{\tau'}$.

3. Compute the product of $B_{\tau'}$ and $B_{\neg\varphi}$. If the product accepts some traces, τ does not satisfy φ and a trace representing the counterexample is raised.

12.4.1 1st step: Rewriting

The first step is the most important one, in fact, it consists in translating a given trace expression to a trace expression representing its *over-approximation*. This phase is necessary because, in general, it is not always possible to translate a trace expression into an equivalent Büchi Automaton; this is due to expressiveness differences. We assume the reader to be familiar with the theory of formal languages and of ω-regular languages, see for example (Hopcroft and Ullman, 1969; Staiger, 1997; Thomas, 1990). A Büchi Automaton recognizes *ω-regular* languages while trace expressions can represent more expressive languages (Chapter 4 for more details) that can be also *ω-context-free* and *ω-context-sensitive*. Consequently, we have to manage all trace expressions which are too expressive (through abstraction). As we introduced in Section 4.2.5, the expressivity of trace expressions is due to the presence of expansive sub terms. In particular, we remind the two kinds of expansive trace expressions we may encounter: the expansive concatenations and the expansive shuffles (Definition 5).

Below we report the pseudocode of a *recursive* implementation of the **rewrite** function (Algorithm 1). This algorithm takes three arguments, that are: the trace expression we want to *over-approximate*, a map of dangerous trace expressions (*dangerous*), a set of already seen trace expressions (*safe* $\cup$ *dangerous*), and it returns the *over-approximation* of the trace expression.

The most interesting part of the algorithm concerns the *expansive* terms recognition. As we have already seen before, an expansive term is a term containing or a concatenation ($\tau_1 \cdot \tau_2$) having a cycle in the head, or a shuffle ($\tau_1 \mid \tau_2$) containing a cycle in its left or right branch (expansive concatenations/shuffles). In order to recognize them, we have to remember the trace expressions we have already visited during the exploration of the trace expression's subtrees differentiating between *safe* and *dangerous* cycles. To achieve this, we maintain in memory the history of all visited states {*safe* $\cup$ *dangerous*} and the history of the *dangerous* states. In this way, we can detect if a cycle is expansive, or not, simply searching inside the *dangerous* map, which must be updated each time we find a new concatenation (or shuffle) operator.

More precisely, the *dangerous* map can be represented as a set of tuples (TExp, Danger) that the algorithm uses to keep track of possibly expansive cycles; where, Danger $\in$ {**maybe**, **true**}, meaning that, if (t, **true**) $\in$ *dangerous*, t is an expansive term and we have to *rewrite* it, while, if (t, **maybe**) $\in$ *dangerous*, we have already seen t in the analysis of TExp and it could be an expansive term because it contains a concatenation or a shuffle subtree. Concluding, if both (t, **true**) and (t, **maybe**) $\notin$ *dangerous*, it means that t is not considered, for now, a dangerous term, because we do not have encounter any concatenation or shuffle subtree inside it.

Before going into the details of the algorithm implementation, it could be useful showing what we obtain when we apply it to a simplified version of the stack example (Section 4.3.1.1).

Example 5. *Considering a stack object, we define the set of correct traces, having only the methods push and pop.*

$[\![\vartheta_{push}]\!] = \{o.push\} \quad [\![\vartheta_{pop}]\!] = \{o.pop\}$

$\tau = \tau_{push} \cdot \tau_{pop}$
$\tau_{push} = \vartheta_{push} : (\tau \vee \epsilon)$
$\tau_{pop} = \vartheta_{pop} : \epsilon$

$rewrite(\tau, \{\}, \{\}) \rightarrow \tau'$
$\tau' = \tau'_{push} \cdot \tau'_{pop}$
$\tau'_{push} = (\vartheta_{push} : \epsilon) \cdot (\tau'_{push} \vee \epsilon)$
$\tau'_{pop} = (\vartheta_{pop} : \epsilon) \cdot (\tau'_{pop} \vee \epsilon)$

Before going on with the presentation of the rewrite algorithm's pseudocode, we can spend some words on this example. Starting from the trace expression τ, we want to obtain a trace expression τ', such that $[\![\tau]\!] \subseteq [\![\tau']\!]$ (τ' is an *over-approximation* of τ). We can easily note that simply deriving the two languages defined by τ and τ'

$$[\![\tau]\!] = \{o.push^n o.pop^n | n \in \mathcal{N}^+\} \cup \{o.push^\omega\}$$

$$[\![\tau']\!] = \{o.push^n o.pop^m | n \in \mathcal{N}^+, m \in \mathcal{N}\} \cup \{o.push^\omega\} \cup \{o.push^n o.pop^\omega | n \in \mathcal{N}^+\}$$

and, since $\{o.push^n o.pop^n | n \in \mathcal{N}^+\} \subseteq \{o.push^n o.pop^m | n \in \mathcal{N}^+, m \in \mathcal{N}\}$, we conclude that $[\![\tau]\!] \subseteq [\![\tau']\!]$. Intuitively, we have just passed from a context-free language to a regular language. With τ we defined that a trace of events containing $o.push$ and $o.pop$ was a correct trace iff the number of $o.push$ was equal to the number of $o.pop$ (with also the possibility of having an infinite number of $o.push$[7]). With τ' we want to reduce instead the expressivity relaxing our constraints. In this specific case, since we are counting and constraining the number of events (making the recognized language at least context-free), when we relax the constraints reducing the expressivity we can simply stop counting the events. Practically speaking, we can just stop forcing $o.push$ and $o.pop$ to have the same cardinality. Removing this constraint it is easy to note that the language we are recognizing now is not context-free anymore, but it is regular. We will show in the the rest of the chapter that these kinds of *over-approximations* allow us to have a more suitable representation of our specifications, in particular, when we are interested in verifying them statically.

In Algorithm 1 we reported the pseudocode of the *rewrite* function. In the following we describe step by step all the cases handled by the algorithm.

[7]Until we see the first $o.pop$ a monitor can not say if the trace is good o not and consequently we accept also the infinite trace of $o.push$.

Algorithm 1 Rewrite function's pseudocode

```
function TExp rewrite(texp, dangerous, already_seen){
 1: if texp = ϵ then
 2:     return ϵ
 3: else if (texp, _) ∈ dangerous then
 4:     remove (texp, maybe) from dangerous
 5:     {texp already visited, we are in a dangerous term} add (texp, true) to dangerous
 6:     return ϵ
 7: else if texp ∈ already_seen then
 8:     return rewritten(texp, already_seen) {texp visited, we are not in dangerous term}
 9: else if texp matches head · tail then
10:     {update the set of texps already visited} already_seen1 = (already_seen ∪ {texp})
11:     {example: {(s1, true), (s2, maybe)} ∪ {s1, s2, s3} =
        {(s1, true), (s2, maybe), (s3, maybe)}} dangerous_head = dangerous ∪ already_seen1
12:     {rewrite the head subterm} new_head = rewrite(head, dangerous_head, already_seen1)
13:     if {t | t ∈ already_seen1 and (t, true) ∈ dangerous_head} ≠ ∅ then
14:         {the concatenation is expansive, thus rewrite the tail subterm} dangerous_tail =
            dangerous ∪ already_seen1
15:         new_tail = rewrite(tail, dangerous_tail, already_seen1)
16:         {create new head and new tail} T1 = new_head · (T1 ∨ ϵ)
17:         T2 = new_tail · (T2 ∨ ϵ)
18:         return T1 · T2
19:     else
20:         remove already_seen1 from dangerous
21:         new_tail = rewrite(tail, dangerous, already_seen1)
22:         return new_head · new_tail {there are no cycles}
23: else if texp matches left | right then
24:     already_seen1 = (already_seen ∪ {texp})
25:     dangerous_left = (dangerous ∪ already_seen1)
26:     dangerous_right = (dangerous ∪ already_seen1)
27:     {dangerous variables refer to different objects} new_left = rewrite(left, dangerous_left,
        already_seen1)
28:     new_right = rewrite(right, dangerous_right, already_seen1)
29:     dangerous = dangerous_left ∪ dangerous_right
30:     t = {t' | t' ∈ already_seen1 and (t', true) ∈ dangerous}
31:     if t ≠ ∅ then
32:         T1 = new_left · T1
33:         T2 = new_right · T2
34:         return T1 | T2
35:     else
36:         return new_left | new_right
37: else if texp matches left ∨ right then
38:     {(the same for ∧)} already_seen1 = copy(already_seen ∪ {texp})
39:     new_left = rewrite(left, dangerous, already_seen1)
40:     new_right = rewrite(right, dangerous, already_seen1)
41:     return new_left ∨ new_right {(the same for ∧)}
42: else if texp matches prefix : body then
43:     {(the same for ≫)} already_seen1 = copy(already_seen ∪ {texp})
44:     new_body = rewrite(body, dangerous, already_seen1)
45:     return prefix : new_body {(the same for ≫)}
}
```

The first case (line 1) tackled is when the trace expression *texp* is the empty trace ϵ. This is the simplest case because we do not have to rewrite anything and we just return the empty trace ϵ.

The second case (line 3) handles the presence of dangerous cycles. If *texp* belongs already to the *dangerous* set[8], it means that *texp* is a cyclic term (since it is in *dangerous* we have already seen it), and it is a subtree of the head of a concatenation term or of a shuffle term. Consequently, *texp* is an expansive term and we have to update the *dangerous* set (now we know it is **true**). After that, we return ϵ (why ϵ will be clear in the fourth and fifth cases).

The third case (line 7) considers the safe cycles. We are not inside the head of a concatenation nor a shuffle, otherwise *texp* would have been inside the dangerous set (second case). Consequently, if *texp* belongs to the set of the *already_seen* trace expressions, it means that we have found a cycle that is not dangerous. We can not return directly *texp*, because *texp* is the old version that can be expansive. Instead, we want to return the new rewritten version of *texp*. In all the cases of the algorithm, each time we update the set of *already_seen* trace expressions, we also implicitly add the rewritten version of the trace expression[9]. In this way, when we encounter a safe cycle we can just return the term that will contain the non-expansive rewritten version of *texp*. We can achieve this using a function called *rewritten* that, given a trace expression and a set of *already_seen* trace expressions, returns the rewritten version.

The fourth case (line 9) is one of the complex cases. Here we handle the concatenation terms. The first thing we have to do is to update the set of *already_seen* trace expressions adding *texp* (the current one). After that, differently from the other simple cases, we have to update also the *dangerous* set, merging it with the set of *already_seen* states. Since now we are approaching to analyze the head of the concatenation, all the terms we have already encounter can cause an expansion of our term, because *texp* is a subtree of each one of them. First of all, we need to analyze the head of the concatenation, if we find a cycle, we handle it as a bad one (second case) and not as a good one (first case). This is totally derived from the definition of expansive concatenation (Definition 5). If we find a dangerous cycle inside the head (the if statement at line 13), we have to rewrite first the tail, and after that, we can construct the new trace expression corresponding to the *over-approximation* of our expansive concatenation. We can obtain that concatenating the new head with the new tail (line 18). Naturally, we have also the good scenario where our concatenation is not expansive, and this can be derived from the absence of dangerous cycles inside the head of the concatenation. In this scenario, we do not change the structure of the concatenation and we limit to concatenate the rewritten head with the rewritten tail. Even though the concatenation is not expansive, there might be expansive terms inside the head or the tail that have been rewritten from the rewrite function (expansive terms not influencing our

[8] We do not care if it is **maybe** or **true**, because now force it to be **true**.

[9] Even though it is not ready, we can add a reference to the term that will be unified with it (in SWI-Prolog, on which the algorithm is actually implemented, we can easily obtain this using free variables).

concatenation because we have checked it at line 13). Before going on with the fifth case, it is time to motivate why at the second step we returned ϵ. In order to understand this, it is enough to see what we do with the rewritten head returned by the function and saved into the variable *new_head*. We create a new term (line 16) concatenating *new_head* with itself. If we think about the original head of the concatenation, where now we have an ϵ, before we had a cycle that made the concatenation expansive. This cycle allowed the concatenation to accumulate a new tail and restart consuming a new head (and so on). If we remove this cycle, we have to simulate the same behavior without the accumulation of the tail. To do so, we can substitute the cycle with a ϵ and combine the new head so generated with itself. In this way we obtain a cycle on the content of the head, as we were doing before but without the accumulation of the tail. To simulate this accumulation we can just do the same thing for the tail, we concatenate the *new_tail* with itself and we conclude concatenating the two terms so generated (line 18). In this way, before we could consume n time the head concatenating with the expansion of the term n times the tail, now, we simply consume a certain number of times n the head without accumulating and, after that, we consume a certain number of times m the tail (where n and m might be different).

The fifth case (line 23) is the other complex case and it is very similar to the fourth one. Here we handle the shuffle terms. Also in this case, we first update the set of *already_seen* terms and then we update the *dangerous* set considering all the super terms of the shuffle (the terms that have *texp* as subtree). After that, we check if we have found a bad cycle inside the left or the right operand, and if so, we rewrite the shuffle using the two new rewritten versions obtained from the two function calls. As it was for the concatenation term, also here we might have found a good shuffle term; consequently, we just construct the new trace expression combining the two rewritten operand terms (left and right) with the shuffle operator. This is totally derived from the definition of expansive shuffle (Definition 5).

In both the fourth and fifth cases, we expect that, if the subtrees of *texp* are not dangerous and do not contain any other expansive terms, *texp* is rewritten into itself.

The sixth case (line 37) handles the union terms (equivalently also the intersection terms). If *texp* is an union term, we simply apply the rewrite function to its operands and we combine the results so obtained using the union operator. Since the union operator does not introduce expansivity *per se*, we do not have to update the *dangerous* set (only concatenations and shuffles introduce expansivity, Definition 5).

The seventh case (line 42) handles the prefix terms (equivalently also the filter terms). If *texp* is a filter term, we simply apply the rewrite function to the remaining of the trace expression. The result will be combined again with the prefix operator, and then returned.

Removing the expansive subtrees (subterms) from the trace expression, we obtain a spurious solution, namely a solution representing a bigger set of

traces; this is entirely due to the fact that only using expansive terms, as cyclic concatenations and shuffles, trace expressions are able to recognize languages more expressive than regular. But, having a less expressive trace expression (the *over-approximation*) we can translate it to an equivalent Büchi Automaton (second step of the algorithm).

Before going on with the presentation of the second step that presents how to translate the rewritten trace expression to its equivalent Büchi Automaton, we have to spend more words on the correctness of the rewrite algorithm. In particular, we have to show that the trace expression returned by it is actually an *over-approximation*. In order to show this, we have to think about what we are trying to do with the rewrite function, that is to remove all the expansive subtrees from the trace expression. Consequently, we can show the correctness of our approach through two steps:

1. First of all, we have to show that given a trace expression τ, the rewrite algorithm removes all expansive subtrees from it returning the corresponding trace expression τ', after that,

2. we have to show that such trace expression τ' is an *over-approximation* of τ, meaning that $[\![\tau]\!] \subseteq [\![\tau']\!]$.

In order to prove the rewrite algorithm removes all the expansive subtrees we have to recall briefly which kind of expansive subtrees we can have, that are *expansive concatenations* and *expansive shuffles* (Definition 5). But, what actually makes a concatenation expansive? As we have already seen previously, a concatenation is expansive if the entire concatenation is a subtree of its head. For instance, $\tau = (\vartheta_{push}{:}(\tau \vee \epsilon)) \cdot (\vartheta_{pop}{:}\epsilon)$ is an expansive concatenation because τ is inside the head. In the same way, we can recall when a shuffle is expansive, namely when the entire shuffle is inside the left or the right operand. For instance, $\tau = (\vartheta_{push}{:}\tau)|(\vartheta_{pop}{:}\epsilon)$ is an expansive shuffle because τ is inside the left operand (left subtree of the shuffle).

Now we show that, starting from an expansive concatenation, the rewrite algorithm removes all the expansive subtrees (the reasoning about the shuffle is almost the same). Recalling the algorithm's pseudocode presented in Algorithm 1, when τ is a concatenation we fall in the fourth case (line 9). The first thing that the algorithm does is updating the set of *already_seen* terms, since now we are analyzing τ, we will add τ to it. After that, since we are in a concatenation, we also update the *dangerous* set adding τ with value **maybe** (we do not know if it is really dangerous or not yet). Now we can call recursively the rewrite function on the head of the concatenation, passing the new updated sets. If the concatenation is expansive, that means we will encounter it again during the evaluation of the head. Since before going into the head we updated the *dangerous* set, if we encounter the concatenation during the evaluation of the head, we fall into the second step (line 3), because τ belongs to the *dangerous* set (with value **maybe**). At this point we know that we are analyzing an expansive subtree for sure. Thus, we can stop analysing it and we can just return back ϵ. The new head is obtained simply concatenating the head

returned by the recursive call of the rewrite function, with itself (line 16). The new tail is also obtained in the same way (line 17). Consequently, the new concatenation is just the concatenation of the new head with the new tail (line 18). As already anticipated when we commented the pseudocode presented in Algorithm 1, combining the new head with the new tail we are simulating the previous behavior without counting the events. The rest of the term structure is unchanged. Considering again the example sketched before, we do not count *o.push* and *o.pop* anymore, but we preserve their order. For instance, it will never happen that *o.push* is observed after *o.pop* (and so on).

For the expansive shuffle terms is almost the same. We update the *dangerous* set, we evaluate the left and right operands and once the rewritten versions are returned, we simply concatenate the left operand with itself (the same for the right operand) and we construct the result as the shuffle of the two terms so constructed.

Now that we have shown informally that given a trace expression τ the rewrite function remove the expansive cycles returning the rewritten version τ'; we have to show that τ' is an *over-approximation* of τ.

If τ is an expansive trace expression, it means that contains expansive concatenations or expansive shuffles inside. When the rewrite function finds an expansive concatenation, it rewrites it simulating its behavior without the accumulation of the tail. As we have shown before, we obtain in this way a new non-expansive concatenation where the number of events generated consuming the head is independent from the number of events generated consuming the tail. But if it is so, it means that the new concatenation generates a bigger set of events, namely $[\![\tau]\!] \subseteq [\![\tau']\!]$. Thus, we can conclude that τ' is an *over-approximation* of τ (a similar reasoning can be done also for expansive shuffle terms).

Now that we have finished to present the first step of the algorithm and we have obtained the *over-approximation* of the trace expression, we can show how we can translate such trace expression into an equivalent Büchi Automaton.

12.4.2 2nd step: Translation

The second step consists in the translation of the trace expression, which was previously rewritten, in an equivalent Büchi Automaton recognizing the same language (set of traces).

Since this trace expression does not contain expansive terms (we removed them in the first step), we can translate it directly simulating the δ transition relation.

Given a non-expansive trace expression τ:

1. create a Büchi Automaton B_τ with an initial state T_0 associated to τ.
2. create a queue of couples Q containing the couple (τ, T_0).
3. extract the first couple (τ_i, T_i) from Q, knowing that the set of events which belong to an event types is finite, for each event e s.t $\tau_i \xrightarrow{e} \tau_i'$:

- if $\tau_i' = \epsilon$, considering ψ a special event which does not belong to our set of possible events, we create a new state T_ψ and we add $T_i \xrightarrow{\psi} T_\psi$, $T_\psi \xrightarrow{\psi} T_\psi$ and we make T_i final (if T_ψ already exists, we add only the edge from T_i to it).
- otherwise if τ_i' has already been seen before, we retrieve the corresponding automaton's state T_r and we add $T_i \xrightarrow{e} T_r$ and we make T_r final.
- otherwise, we create a new state T'_i associated to τ_i' and we add $T_i \xrightarrow{e} T_i'$.

4. if Q is not empty, we restart from the point 3.

At the end of this process we obtain the Büchi Automaton B_τ equivalent to τ (see Theorem 7).

The ψ special event is used to make acceptable by the Büchi Automaton also the finite traces. In this way, if τ recognizes the finite trace $\sigma = abcd$, the corresponding Büchi Automaton will recognize the trace $u = abcd\,\psi^\omega$. It is important that the ψ event must not belong to the set of handled events in order to avoid *False Negative* results, where the ψ could make a LTL property erroneously satisfied by u.

Example 6. *Considering the same τ' obtained at the end of Example 5, we create the following Büchi Automaton $B_{\tau'}$:*

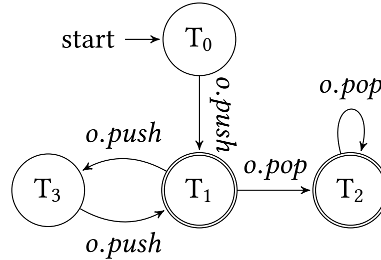

FIGURE 12.1. Büchi Automaton $B_{\tau'}$

In Example 6, the trace expression τ' and the corresponding Büchi Automaton represent a superset of the traces recognized by the initial trace expression τ. In particular, we lose the ability to bind the number of *o.push* with the number of *o.pop* events. In fact, we can only represent traces where we have a number n_{push} of *o.push* before a number n_{pop} of *o.pop*, with $n_{push} \neq n_{pop}$ possibly. Thus, we have conserved only the causality between *o.push* and *o.pop*.

Lemma 3. *Given a trace expression τ, if τ is non-expansive then the set of trace expressions that can be obtained starting from τ in an arbitrary number of steps is finite.*

Proof. If τ is *non-expansive* we do not fall back in the cases like that showed in Example 1 and Example 2 where we have a trace expression generating an infinite set of new states (through term expansion). Removing all the expansive concatenations and shuffles, we have trace expressions that in an arbitrary number of steps become ϵ (we can terminate because we have no more steps to do) or become a trace expression already seen (we can terminate because we have already visited this trace expression). This is easy to note considering the operational semantics of trace expressions (see Figure 4.1). □

Theorem 7. *Let τ be a non-expansive trace expression, the Büchi Automaton B_τ obtained simulating the δ transition relation is equivalent to τ and it is always computable (Lemma 3), thus*

$$\forall_{u \text{ infinite trace}}. u \in [\![\tau']\!] \iff B_{\tau'} \text{ accepts } u$$
$$\forall_{\sigma \text{ finite trace}}. \sigma \in [\![\tau']\!] \iff B_{\tau'} \text{ accepts } \sigma \cdot \psi^{\omega}.$$

Proof. It follows directly from the Büchi Automaton $\mathrm{B}_{\tau'}$ construction. □

12.4.3 3rd step: Product

The last step consists in the real model checking phase.

In the previous steps, starting from a trace expression τ representing our model we generated its *over-approximation* τ' to have a model expressive as a Büchi Automaton (*1st step*), after that we translated τ' to its corresponding Büchi Automaton representation $\mathrm{B}_{\tau'}$ (*2nd step*). The only step left to complete is check if a LTL property φ is satisfied by the model $\mathrm{B}_{\tau'}$, or not.

As we presented in Section 3.6.3, we can follow an *Automata-Based Model Checking* approach computing the product $\mathrm{B}_{\tau' \times \neg\varphi}$ between $\mathrm{B}_{\tau'}$ and $\mathrm{B}_{\neg\varphi}$. Once $\mathrm{B}_{\tau' \times \neg\varphi}$ is created, we can test its emptiness, and if it is not we have found a counterexample and we can conclude that τ' does not satisfy φ.

To fill the gap we have to remind that τ and τ' are related.

Observation 1. $[\![\tau]\!] \subseteq [\![\tau']\!]$, τ' over-approximation *of* τ.

Since τ' is an *over-approximation* of τ and $\mathrm{B}_{\tau'}$ is its automata representation, if we do not find any traces (counterexample) which belongs to $\mathrm{B}_{\tau' \times \neg\varphi}$, we can deduce:

$$\text{(from } \textit{Theorem 7}\text{)}$$
$$\nexists_{u \in \mathrm{B}_{\tau'}}. u \text{ satisfies } \neg\varphi \iff \nexists_{u \in [\![\tau']\!]}. u \text{ satisfies } \neg\varphi$$
$$\text{(from } \textit{Observation 1}\text{)}$$
$$\nexists_{u \in [\![\tau']\!]}. u \text{ satisfies } \neg\varphi \implies \nexists_{u \in [\![\tau]\!]}. u \text{ satisfies } \neg\varphi$$

In this way, we can conclude that, if the *over-approximation* τ' satisfies φ (no counterexample has been found) then also τ satisfies φ. From Observation 1 we can deduce the implication for only one direction ($\implies$) because, since we are over-approximating our model, there could exist a counterexample trace u which does not satisfy φ s.t $u \in [\![\tau']\!]$ but $u \notin [\![\tau]\!]$ (*False Positive*).

One of the main advantages of following a standardized approach as the *Automata-Based Model Checking* is that, once we have obtained the Büchi Automaton $B_{\tau' \times \neg\varphi}$, we can represent it directly using an existent and well-know programming language as PROMELA inside the popular open-source software verification tool SPIN[10] (Holzmann, 1991, 1997, 2004).

12.5 Implementation and Experiments

In the following we continue to use the stack example used in the rest of the chapter. We show an experiment of use of our SWI-Prolog implementation of the 3-steps algorithm presented in Section 12.4. The code implementing the translation can be downloaded from `http://trace2buchi.altervista.org`.

Inside the SWI-Prolog environment, starting from the trace expression `T` representing our very simple stack, we first rewrite `T` into the corresponding non-expansive abstraction `Tr` (*1st Step*) using the corresponding `rewrite` predicate, then we translate `Tr` into the equivalent Büchi Automaton `Buchi` (2nd Step) using the corresponding `translate` predicate, finally we create a promela file `stack.pml` containing the `Buchi` representation and the LTL property that we want to check (for instance, `eventually pop` (`<>(pop)`)).

```
?- T = ((push:(T\/epsilon))*(pop:epsilon)),                    1
rewrite(T, Tr), translate(Tr, Buchi),                          2
write_promela_file(stack, Buchi, '(<>(pop))').                 3
```

The previous SWI-Prolog implementation corresponds to the first two steps of our 3-steps algorithm (Section 12.4.1 and 12.4.2). This execution brings to the creation of the `stack.pml` file.

In the following we report the `stack.pml` content so obtained.

```
bool epsilon = 0;                                                          1
bool pop = 0;                                                              2
bool push = 0;                                                             3
                                                                           4
active proctype stack() {                                                  5
  S0_init:                                                                 6
    if                                                                     7
      :: skip -> d_step { push=1; epsilon=0; pop=0 } goto accept_S1        8
    fi                                                                     9
  accept_S1:                                                               10
    if                                                                     11
      :: skip -> d_step { push=1; epsilon=0; pop=0 } goto S2               12
    fi                                                                     13
  S2:                                                                      14
    if                                                                     15
      :: skip -> d_step { pop=1; epsilon=0; push=0 } goto accept_S3        16
      :: skip -> d_step { push=1; epsilon=0; pop=0 } goto accept_S1        17
    fi                                                                     18
  accept_S3:                                                               19
    if                                                                     20
      :: skip -> d_step { epsilon=1; pop=0; push=0 } goto accept_Seps      21
```

[10] `http://spinroot.com/spin/`

```
      :: skip -> d_step { pop=1; epsilon=0; push=0 } goto accept_S3   22
    fi                                                                 23
  accept_Seps:                                                         24
    if                                                                 25
      :: skip -> d_step { epsilon=1; pop=0; push=0 } goto accept_Seps  26
    fi                                                                 27
}                                                                      28
ltl { (<>(pop)) }                                                      29
```

This file is the promela code corresponding to the Büchi Automaton presented in Figure 6.

The states in promela correspond to the Büchi Automaton states; for instance, `S0_init` corresponds to T_0 in Figure 6 (and similarly for all the other states involved). Also the Büchi Automaton transitions are translated into the promela code, in particular using the `d_step` construct. For instance, the *o.push* transition from T_0 to T_1 is translated into `d_step{push=1;epsilon=0;pop=0}` from `S0_init` to `accept_S1`. In Figure 6, T_1 was an accepting state and so it is in promela. The `accept_*` pattern is the one used in promela to represent accepting states.

Finally, having the Büchi Automaton representation of the trace expression resulting from the *2nd Step*, we can do the product between `Buchi` and the LTL property (Section 12.4.3) directly inside the SPIN model checker (*3rd Step*).

First of all, using the `stack.pml` file, we generate a verifier (model checker) for the specification.

In the shell we write:

```
~$ spin -a stack.pml   1
```

The output generated is a C file, named `pan.c`, that can be compiled to produce an executable verifier.

In the shell we compile the `pan.c` file obtaining the executable corresponding to our verifier.

```
~$ gcc pan.c -o stack_verifier   1
```

Then, we can run the verifier. Since the LTL property that we want to check is a liveness property, we have to add the `-a` flag in order to find acceptance cycles (violations are infinite executions).

```
~$ ./stack_verifier -a                                1
 pan:1: acceptance cycle (at depth 4)                 2
 ...                                                  3
 State-vector 28 byte, depth reached 12, errors: 1    4
         8 states, stored (10 visited)                5
         0 states, matched                            6
        10 transitions (= visited+matched)            7
         0 atomic steps                               8
 ...                                                  9
```

As we can see, the verifier finds an acceptance cycle (that is a violation of the LTL property). Using the `-r` flag we can read and execute the trail file (generated in the previous step) containing the counterexample trace.

```
~$ ./stack_verifier -r   1
MSC: ~G 4                2
```

```
  1:     proc  0 (ltl_0) stack.pml:4 (state 3)   [(!((pop)))]
  2:     proc  1 (stack) stack.pml:8 (state 7)   [(1)]
  3:     proc  0 (ltl_0) stack.pml:4 (state 3)   [(!((pop)))]
  4:     proc  1 (stack) stack.pml:8 (state 5)   [D_STEP8]
<<<<<START OF CYCLE>>>>>
  5:     proc  0 (ltl_0) stack.pml:4 (state 3)   [(!((pop)))]
  6:     proc  1 (stack) stack.pml:12 (state 15) [(1)]
  7:     proc  0 (ltl_0) stack.pml:4 (state 3)   [(!((pop)))]
  8:     proc  1 (stack) stack.pml:12 (state 13) [D_STEP12]
  9:     proc  0 (ltl_0) stack.pml:4 (state 3)   [(!((pop)))]
 10:     proc  1 (stack) stack.pml:16 (state 29) [(1)]
 11:     proc  0 (ltl_0) stack.pml:4 (state 3)   [(!((pop)))]
 12:     proc  1 (stack) stack.pml:17 (state 27) [D_STEP17]
spin: trail ends after 12 steps
#processes 2:
 12:     proc 0 (ltl_0)  stack.pml:4 (state  3) (invalid end state)
                (!((pop)))
 12:     proc 1 (stack)  stack.pml:12 (state 15) (invalid end state)
                (1)
global vars:
        bit    pop:     0
local vars proc 1 (stack):
```

This cycle corresponds to the infinite trace containing only the `push` event. Consequently, it is not true that our specification recognizes only traces where `<>(pop)` is satisfied. In fact, $o.push^{\omega} \in [\![\tau]\!]$ (that is exactly the trace causing the generation of the trail file reported above).

We can check another LTL property. This time a safety property, for instance, `globally push([](push))`. As before, we execute the predicates corresponding to the two first steps of the algorithm directly inside SWI-Prolog (passing the new LTL property as argument). After that we can compile the new `pan.c` file generated with the `-DSAFETY` flag. Since the LTL formula is a safety property we need to search for assertion violations (violations are finite executions).

Finally we can run the model checker obtaining the following result.

```
~$ ./stack_verifier
pan:1: assertion violated  !( !((push))) (at depth 12)
...
State-vector 20 byte, depth reached 12, errors: 1
        7 states, stored
        0 states, matched
        7 transitions (= stored+matched)
        0 atomic steps
...
```

Also here, we find a counterexample that violates our LTL property. This time, since we are verifying a safety property, we search for a state violating an assertion.

As we did for the previous experiment, also here we can see the trail file generated by the model checker.

```
~$ ./stack_verifier -r
MSC: ~G 3
  1:     proc  0 (ltl_0) stack.pml:3 (state 6)   [(1)]
  2:     proc  1 (stack) stack.pml:8 (state 7)   [(1)]
  3:     proc  0 (ltl_0) stack.pml:3 (state 6)   [(1)]
```

```
  4:     proc  1 (stack) stack.pml:8 (state 5)   [D_STEP8]             6
  5:     proc  0 (ltl_0) stack.pml:3 (state 6)   [(1)]                 7
  6:     proc  1 (stack) stack.pml:12 (state 15) [(1)]                 8
  7:     proc  0 (ltl_0) stack.pml:3 (state 6)   [(1)]                 9
  8:     proc  1 (stack) stack.pml:12 (state 13) [D_STEP12]            10
  9:     proc  0 (ltl_0) stack.pml:3 (state 6)   [(1)]                 11
 10:     proc  1 (stack) stack.pml:16 (state 29) [(1)]                 12
 11:     proc  0 (ltl_0) stack.pml:3 (state 6)   [(1)]                 13
 12:     proc  1 (stack) stack.pml:16 (state 21) [D_STEP16]            14
pan:1: assertion violated  !( !((push))) (at depth 13)                 15
spin: trail ends after 13 steps                                        16
#processes 2:                                                          17
 13:     proc 0 (ltl_0)  stack.pml:3 (state  6) (invalid end state)    18
                 (!((push)))                                           19
                 (1)                                                   20
 13:     proc 1 (stack)  stack.pml:21 (state 43) (invalid end state)   21
                 (1)                                                   22
                 (1)                                                   23
global vars:                                                           24
         bit    push:    0                                             25
local vars proc 1 (stack):                                             26
```

Differently from the previous experiment, we do not search for cycles, because we are verifying a safety property, but we search for assertion violations. As we can see in the trail file generated, we find a violation after 13 steps, when our property is violated since `push` is not satisfied anymore (for instance, $\{o.push\ o.push\ o.pop\ o.pop\} \in [\![\tau]\!]$ and $\{o.push\ o.push\ o.pop\ o.pop\}$ does not satisfy `[](push)`).

12.6 Discussion

In this chapter, we showed how to use a standard static approach to verify a rich formalism used to generate monitors for runtime verification of object-oriented programs. By verifying LTL properties statically we obtain two main advantages: (1) we can check if the specification of our monitor is coherent with our intentions, and (2) the system monitored by the verified monitor satisfies the same LTL properties, as long as it is consistent with the specification.

The first two steps of the algorithm presented in Section 12.4 are implemented in SWI-Prolog, while the last step (the Büchi Automaton product) is implemented using the SPIN model checker. The Büchi Automaton B_τ generated in the second step and the LTL property φ we want to verify are both compiled to PROMELA language.

13 *Recognising Assumption Violations in Autonomous Systems Verification*

"Your assumptions are your windows on the world.
Scrub them off every once in a while,
or the light won't come in."
- Isaac Asimov

When applying formal verification to a system that interacts with the real world we must use a model *of the environment. This model represents an* abstraction *of the actual environment, but is necessarily incomplete and hence presents an issue for system verification. If the actual environment matches the model, then the verification is correct; however, if the environment falls outside the abstraction captured by the model, then we cannot guarantee that the system is well-behaved. A solution to this problem consists in exploiting the model of the environment for statically verifying the system's behaviour and, if the verification succeeds, using it also for validating the model against the real environment via runtime verification. The chapter discusses this approach and demonstrates its feasibility by presenting its implementation on top of a framework integrating the Agent Java PathFinder model checker. Trace expressions are used to model the environment for both static formal verification and runtime verification.*

The contents of this chapter are published in (Ferrando et al., 2018a,b) and derive from a joint collaboration with the Autonomy and Verification Laboratory[1] *at the University of Liverpool*[2]

[1] `http://cgi.csc.liv.ac.uk/~matt/AVLab/`
[2] `https://www.liverpool.ac.uk`

13.1 Introduction

Static formal verification of autonomous systems that interact with the real world requires a model of the world to successfully accomplish the verification process. (Dennis et al., 2014) recommends using the simplest environment model, in which any combination of the environmental predicates that correspond to possible perceptions of the autonomous system is possible.

Consider an intelligent cruise control in an autonomous vehicle that can perceive the environmental predicates:

- `safe`, meaning it is safe to accelerate,
- `at_speed_limit`, meaning that the vehicle reached its speed limit,
- `driver_brakes` and `driver_accelerates`, meaning that the driver is braking/accelerating.

In order to formally verify the behaviour of the cruise control agent, we might randomly supply subsets of

{`safe`, `at_speed_limit`, `driver_brakes`, `driver_accelerates`}

where the generation of each subset causes branching in the state space exploration during verification so that, ultimately, all possible combinations are explored.

This model is an *unstructured abstraction* of the world, as it makes no specific assumptions about the world behaviour and deals only with the possible incoming perceptions that the system may react to. Unstructured abstractions obviously lead to significant **state space explosion**.

The state space explosion problem can be addressed by making assumptions about the environment.

For instance, we might assume that a car cannot both brake and accelerate at the same time: subsets of environmental predicates containing both `driver_brakes` and `driver_accelerates` should not be supplied to the agent during the static verification stage, as they do not correspond to situations that we believe likely in the actual environment. This *structured abstraction* of the world is grounded on assumptions that help prune the possible perceptions and hence control state space explosion.

Structured abstractions have advantages over unstructured ones, provided that the assumptions they rely on are correct. Let us suppose that the cruise control system crashes if the driver is accelerating and braking at the same time. If the subsets of environmental predicates generated to verify it never contain both `driver_brakes` and `driver_accelerates`, then the static formal verification succeeds but if one real driver, for whatever reason, operates both the acceleration and brake pedals at the same time, the real system crashes!

In this chapter, we propose an approach for exploiting the advantages of structured abstractions, while mitigating their risks. Our proposal consists in modelling the structured abstraction in a formalism that can be used both

for statically verifying the autonomous system's behaviour via model checking and for validating the model against the real environment by means of runtime verification (RV). If performed during a testing stage, RV of the actual environment against its structured abstraction allows the developer to identify situations not foreseen in the initial assumptions. He/she can revise them, generate a new structured abstraction, re-verify it via model checking, re-validate it via RV once again, reaching in the end a "safe" abstraction. If RV takes place after system deployment and assumption violations are detected, mechanisms for handing control to a human, a failsafe system, or for performing ad hoc reasoning should be invoked.

To demonstrate the feasibility of the proposed approach, we implemented it on top of the MCAPL framework (Dennis, 2018; Dennis et al., 2012) (which provides a model-checker for rational agents, see Section 3.7 for further details) using trace expressions (Chapter 4) as the single formalism to generate both the environment model and the runtime monitor.

We choose trace expressions instead of more widely used formalisms for model checking like LTL because of their expressive power. In Section 4.5, we demonstrated that trace expressions are able to express and verify sets of traces that are context-free. When working in a RV scenario, trace expressions are more expressive than LTL (Section 4.5). In this chapter, we keep the presentation as simple as possible and do not stress the potential of such expressive power. However, this power opens up interesting scenarios discussed in Section 13.5 and in Chapter 17.

13.2 State of the art

The growing popularity of model checking in industry is due to the possibility of transforming domain-specific input models familiar to the developers into "under the hood models" invisible to them and amenable to model checking using existing techniques (Merwe, Merwe, and Visser, 2012). The idea behind this work is similar: we use trace expressions as the front-end formalism suitable for modeling behaviour patterns in systems made up of autonomous entities and we transform them into under the hood models suitable for both model checking and runtime verification (RV). The main difference is that trace expressions are not domain-specific, and although initially devised for modeling protocols in multiagent system, they have been successfully adopted for specifying different kinds of behavioural patterns, including interactions among objects in Java-like programs (Chapter 6 and (Ancona et al., 2017a)) and Internet of Things applications developed with Node.js[3] (Ancona et al., 2017b). This is both a strength and a weakness: a customised formalism for different domains would make it more usable by domain experts, at the cost of some loss in generality.

"Enabling sufficiently precise yet tractable verification" with models – be them explicit or under the hood – of the real environment is a main issue

[3]https://nodejs.org/it/

(Tkachuk, Dwyer, and Pasareanu, 2003). Developing "safe" environment models[4] for model checking that are sufficiently precise to enable effective reasoning yet not so over-restrictive that they mask faulty system behaviours has been understood as a significant challenge since the early 2000s (Penix et al., 2000). The Bandera Environment Generator (Tkachuk, Dwyer, and Pasareanu, 2003) is a toolset that automates the generation of environments to provide a restricted form of modular model checking of Java programs. Although the addressed problem is the same as ours, the approach is different. We do not automatically generate "safe by construction" trace expressions starting from observations of the environment. Rather, we manually design and implement a trace expression encoding our assumptions and validate it against the real environment to empirically show that it is "safe". Although our approach requires a more accurate design stage and more manual work, it can be applied to any system and environment; the automatic generation of the environment model is instead inherently domain-dependent, and the Bandera Environment Generator is in fact customized for model checking Java programs. A form of semi-automatic environment model generation would be possible if we were able to synthesise trace expressions from event traces collected from previous observations[5]. The approach of Dhaussy et al. (Dhaussy, Roger, and Boniol, 2011) is closer to ours; the state space explosion is mitigated with requirements relative to scenarios which are verified instead of the full environment. In that work the context – corresponding to our structured abstraction – is modelled with the domain-specific Context Description Language, CDL. The main difference is that CDL is less expressive than trace expressions (recursion and concatenation are not supported), and no methodology for checking the CDL specification against the real environment is discussed. In a similar way, in (Desai, Dreossi, and Seshia, 2017) Desai et al. present a framework to combine model checking and runtime verification for robotic applications. They represent the discrete model of their system using the *P* language (Desai et al., 2013), check the model and extract the assumptions deriving from such abstraction. Despite sharing the same purpose, our work is not committed to any specific case study and trace expressions are more expressive than STL specifications (Maler and Nickovic, 2004) used in (Desai, Dreossi, and Seshia, 2017). Besides CDL, hybrid automata (Alur et al., 2000; Henzinger, 1996) are another widely adopted formalism for precise modelling of the real world. They do not solve the question of whether the model accurately captures the environment, and although RV of cyber-physical systems modelled with hybrid automata is a lively and promising research field (Nguyen et al., 2015; Sistla, Žefran, and Feng, 2012), we are not aware of proposals where the same hybrid automaton model undergoes both a model checking and a RV process, to both formally prove system properties and validate the correctness of the environment model.

Investigation of model checking for MAS dates back to 1998 (Benerecetti, Giunchiglia, and Serafini, 1998) and has continued to generate much follow

[4] "Environment models" and "structured abstractions of the environment" are used as synonyms in this context.

[5] Exploring this issue is in our agenda, but it is not in the scope of this work.

up work, for instance the Model Checking Agent Programming Languages project[6] (Bordini et al., 2006; Dennis et al., 2012), and other work (Lomuscio and Raimondi, 2006; Raimondi and Lomuscio, 2007). Approaches to MAS RV complement these and include the proposals spun off from the SOCS project[7] where the SCIFF computational logic framework (Alberti et al., 2005) is used to provide the semantics of social integrity constraints. To model MAS interaction, expectation-based semantics specifies the links between the observed events and the expected ones, providing a means to test run-time conformance of an actual conversation with respect to a given interaction protocol (Torroni et al., 2009). Similar work has been performed using commitments (Chesani et al., 2009). None of the contributions above tackle the problem of recognising assumption violations in structured abstractions via RV, for model checking autonomous systems immersed in a real environment. This makes the content of this chapter original in the panorama of model checking both "in general", and, more specifically, for autonomous systems and MAS.

13.3 Running Example

The MCAPL framework (Section 3.7) supports model checking of programs in BDI-style languages via the implementation of interpreters for those languages in Java. The framework implements *program model-checking* in which the *actual* program to be verified, not a model of it, is checked, and contains the Agent Java PathFinder (AJPF) model checker which customises the Java PathFinder (JPF) model checker for Java bytecodes[8] (Section 3.7).

BDI-based languages are based on *rational agency* (Section 3.2). We use the "Engineering Autonomous Space Software" (EASS) variant of GWENDOLEN (Dennis, 2017), a language developed for programming agent-based autonomous systems and verifying them in AJPF. EASS assumes an architecture in which the rational agents are partnered with an *abstraction engine* that discretises continuous information from sensors in an explicit fashion (Dennis et al., 2010, 2016).

We adopt the methodology from (Dennis et al., 2014) setting out the formal verification of rational agent components in autonomous systems. This uses model checking to demonstrate that the rational agent always tries to act in line with requirements and never *deliberately* chooses options that lead to states the agent believes to be unsafe.

AUTONOMOUS CRUISE CONTROL. The (slightly simplified) EASS code in Example 7 is for an agent implementing intelligent cruise control in an autonomous vehicle. It uses standard syntactic conventions from BDI agent languages:

- `+!g` indicates the addition of a goal,g;

[6] http://cgi.csc.liv.ac.uk/MCAPL/
[7] http://lia.deis.unibo.it/research/projects/SOCS/
[8] https://babelfish.arc.nasa.gov/trac/jpf

- `+b` indicates the addition of a belief, `b`; and
- `-b` indicates the removal of a belief.

Plans follow the pattern

$$\text{trigger} \; : \; \text{guard} \; \longleftarrow \; \text{body;}$$

The trigger is the addition of a goal or a belief (beliefs may be acquired thanks to the operation of perception or as a result of internal deliberation); the guard states conditions about the agent's beliefs which must be true before the plan can become active; and the body is a stack of *deeds* the agent performs in order to execute the plan. These deeds typically involve the addition or deletion of goals and beliefs, as well as *actions* (e.g. `perf(accelerate)`, meaning "perform the action of accelerating") which indicate code delegated to non-rational parts of the system.

According to the operational semantics of GWENDOLEN (Dennis, 2017), the agent moves through a *reasoning cycle* polling an external environment for perceptions; converting these into beliefs and creating intentions from new beliefs; selecting an intention for consideration; if the intention has no associated plan body, then the agent seeks a plan that matches the trigger event and places the body of this plan on the deed stack; the agent then processes the first deed, and places the intention at the end of the intention queue before again performing perception. As an intention may be suspended while it waits for some belief to become true, we use `*b` to indicate a deed that suspends processing of an intention until `b` is believed. Plan guards are evaluated using Prolog-style reasoning with *reasoning rules* of the form `h :- body` and literals drawn from agent's belief base. Negation is indicated with ~ and its semantics is negation by failure as in Prolog.

Example 7 (Cruise Control Agent)**.** *When the car has a goal to be at the speed limit,* `+! at_speed_limit`*, it can accelerate if it believes it to be safe, that there are no incoming instructions from the human driver, and it does not already believe it is accelerating or is at the speed limit — it does this by removing any belief that it is braking, adding a belief that it is accelerating, performing acceleration, then waiting until it no longer believes it is accelerating. If it does not believe it is safe, believes the driver is accelerating or braking, or believes it is already accelerating, then it waits for the situation to change. If it believes it is at the speed limit, it maintains its speed having achieved its goal (which will be dropped automatically, having been achieved).*

If new beliefs arrive from the environment that the car is at the speed limit, no longer at the speed limit, no longer safe, or the driver has accelerated or braked, then it reacts appropriately. Note that even if the driver is trying to accelerate, the agent only does so if it is safe.

```
:Reasoning Rules:                                                1
can_accelerate :- safe, ~ driver_accelerates,                    2
~ driver_brakes;                                                 3
                                                                 4
```

```
:Initial Goal: at_speed_limit                                   5
                                                                6
:Plans:                                                         7
+! at_speed_limit:                                              8
 {can_accelerate, ~accelerating, ~at_speed_lim}                 9
  ← −braking, +accelerating, perf(accelerate),                  10
       *~accelerating;                                          11
+! at_speed_limit: {~safe} ← *safe;                             12
+! at_speed_limit: {driver_accelerates}                         13
   ← *~driver_accelerates;                                      14
+! at_speed_limit: {driver_brakes}                              15
   ← *~driver_brakes;                                           16
+! at_speed_limit: {accelerating}                               17
   ← *~accelerating;                                            18
+at_speed_lim: {can_accelerate, at_speed_lim}                   19
   ← −accelerating, −braking,                                   20
       perf(maintain_speed);                                    21
−at_speed_lim: {~at_speed_lim}                                  22
   ← +! at_speed_limit;                                         23
−safe: {~driver_brakes, ~safe, ~braking} ←                      24
      −accelerating, +braking, perf(brake);                     25
+driver_accelerates:  {safe, ~driver_brakes,                    26
              driver_accelerates, ~accelerating}                27
   ← +accelerating, −braking, perf(accelerate);                 28
+driver_brakes: {driver_brakes, ~braking} ←                     29
      +braking, −accelerating, perf(brake);                     30
```

The cruise control agent has to be connected to either a physical vehicle or a simulation. Similar EASS agents have been connected to both detailed simulations of ground vehicles and physical vehicles (Dennis et al., 2016; Kamali et al., 2017). Here we will consider embedding the agent within a multi-lane, multi-vehicle motorway (highway) simulation.

The agent is connected to the simulator via a *thin Java environment* that communicates using sockets. The environment reads simulated speeds of the vehicles from the socket and publishes values for acceleration to the socket. The information from sensors is then passed on to an *abstraction engine* that converts it to discrete representations, shared with the rational agent as logical predicates. The rational agent accesses these *shared beliefs* as perceptions.

Previously, the model of the combined behaviour of simulator, thin Java environment, and abstraction engine used for verification was unstructured: all the possible combinations of the shared beliefs were explored. This is where our proposal for modeling structured abstractions as trace expressions and validating them via RV, as well as using them for model checking, comes into play.

13.4 Recognising Assumption Violations

In this section we discuss how trace expressions can be suitably adopted for specifying structured abstractions of the real world for use in AJPF. The idea is generate *both* a suitable Java model for AJPF model checking *and* a runtime monitor from the same trace expression. The monitor can detect if the real (or simulated) environment violates the assumptions used during the static verification.

13.4.1 AJPF Static Formal Verification

The EASS implementation provides a Java class supporting the creation of abstract models. Unstructured abstractions can be created by overriding in a *subclass* its method `add_random_beliefs` which is called when the agent requests an action execution or sleeps. This method should generate a set of beliefs and add them to the environment's *percept base* which the agent then polls. It is assumed this implementation will randomly generate all possible subsets of the shared beliefs relevant to the agent. For static verification, therefore, we want to generate this subclass from our trace expression. In normal operation, EASS abstraction engines communicate with the agent-based reasoning engine (the 'agent') by performing `assert_belief` and `remove_belief` actions. These actions are implemented by Java environments which also connect to sensors and simulators. There are four such actions: `assert_belief(b)` asserts a shared belief for all agents and `remove_belief(b)` removes shared belief b from all agents. `assert_belief(a, b)` and `remove_belief(a, b)` alter the available beliefs for a specific agent `a`. Our runtime monitor needs to observe these events. We are also interested in any action performed by an agent, so our runtime monitor must also observe calls to the `executeAction` method that all EASS environments implement. Figure 13.1 gives an overview of this system. A trace expression τ is used to generate an abstract model in Java used to verify an agent in AJPF (the dotted box on the right of Figure 13.1). Once this verification is successfully completed, the verified agent is used with an abstraction engine, a thin Java environment, and the real world or external simulator. This is shown in the dotted box on the left of Figure 13.1. If, at any point, the monitor observes an inconsistent event we can conclude that the abstraction used during verification was incorrect. Depending upon the development lifecycle stage reached so far different measures will be possible, ranging from refining the trace expression and re-executing the verification-validation steps, to involving a human or a failsafe system in the loop.

13.4.2 Event Types for AJPF Environments

We have identified the assertion and removal of shared beliefs and the performance of actions as the "events of interest" in our Java environments. Our runtime monitor receives notification of all actions in the environment as events. It is possible to flexibly create a number of different event types on top of this structure, so we define a number of default event types that are widely used throughout the system. The following event types model events involving shared beliefs:

- $bel(b)$, where $e \in bel(b)$ iff $e = assert_belief(b)$;
- $not_bel(b)$, where $e \in not_bel(b)$ iff $e = remove_belief(b)$;

We coalesce these as event set $\mathcal{E}_b$ and define event types:

- $action(any_action)$ where $e \in action(any_action)$ iff $e \notin \mathcal{E}_b$

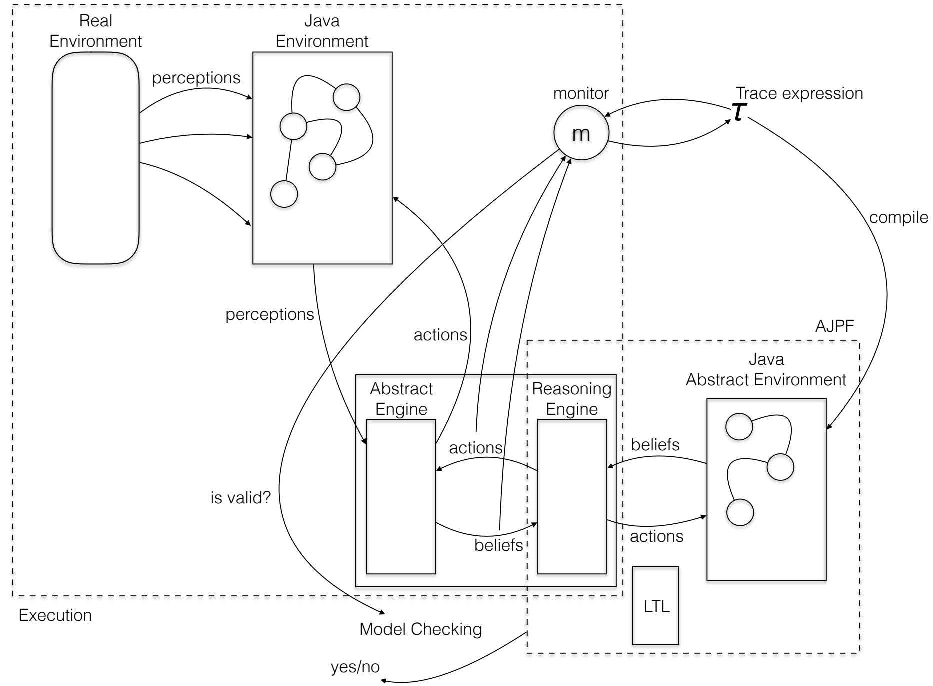

FIGURE 13.1. General view.

- *not_action* where $e \in not_action$ iff $e \in \mathcal{E}_b$
- *action*(A) where $e \in action(A)$ iff $e \notin \mathcal{E}_b$ and $e = A$

Abstract models in AJPF can be represented as automata, with unstructured abstractions representing their most general form. The automaton states can be divided into two parts: *initial beliefs* and *actions*. Initial Beliefs represents all the shared beliefs that may be asserted before the system starts executing. After an action is performed, more shared beliefs may be asserted. In the unstructured abstractions used by the "standard" AJPF system the initial beliefs, and the beliefs after each action, were generated at random.

Any structured abstraction will be one that places constraints upon the possible transitions in the automaton. We represent an abstract model of the real world as a set of possibly cyclic trace expressions modelled in Prolog. The basic structure of the Prolog code is given in Figure 13.2. We abuse regular expression syntax: as parentheses are used for grouping in trace expressions, we adopt [and] to represent groupings within a regular expression; similarly, since | is a trace expression operator, we use || to indicate alternatives within the regular expression. Here, $e?$ indicates zero or one occurrences of the element e. As we use Prolog, variables are represented by terms starting with an upper case letter (e.g., $Action_i$) and constants are represented by terms starting with a lower case letter (e.g., b_i, $action_i$)). $\Big|_{i=1}^{n}$ indicates one or more trace expressions composed via the trace expression shuffle operator, |. Similarly, $\bigvee_{i=1}^{n}$ composes expressions using $\vee$ and $\bigwedge_{i=1}^{n}$ composes expressions using $\wedge$. Variables with the same name will be unified. Occurrences of Pre in (14.1) and (14.2) are intended

$$Protocol = Pre \cdot (Cyclic\ [\ \wedge\ Constrs]?) \tag{13.1}$$

$$Pre = [\ \Big|_{i=1}^{n} bel(b_i){:}\epsilon]\ \|\ [not_action : Pre \vee \epsilon] \tag{13.2}$$

$$Cyclic = SingleStep \cdot Cyclic \tag{13.3}$$

$$SingleStep = \bigvee_{i=1}^{m} Action_i \cdot AddBelEv \tag{13.4}$$

$$AddBelEv = not_action : AddBelEv \vee \epsilon \tag{13.5}$$

$$Action_i = action(action_i){:}ProtocolBel \tag{13.6}$$

$$ProtocolBel = \Big|_{i=1}^{k} (bel(b_i){:}\epsilon \vee not_bel(b_i){:}\epsilon \vee \epsilon) \tag{13.7}$$

Figure 13.2. Trace expression template for generating abstract environments.

to unify, and the variable names used in these positions in any instantiation of this template should be the same.

The template in Figure 13.2 represents an unstructured abstraction in which any subset of the beliefs, b_i in (14.7) can occur after an action. *Protocol* (14.1) is the main body of our trace expression. *Pre* (14.2) represents all events that can be generated before the first action of an agent. *Cyclic* (14.3) is the trace expression that describes the behaviour once the agent starts performing actions. *SingleStep* (14.4) represents a single action step. It is the union of the trace expressions that describe the possible results of each action the agent may take followed by *AddBelEv* which describes additional belief events after the immediate results of the action – for instance if the agent sleeps and other agents are acting. *Action* (14.6) consists of an action event followed by *ProtocolBel* (14.7) which describes the possible belief events. Any given belief, b_i may appear in the shared belief base ($bel(belief_i)$), disappear ($not_bel(belief_i)$) or its status may be unchanged (ϵ).

Figure 13.2 contains an optional variable *Constrs*. If present this provides *constraints* that structure the abstraction. The template for constraints is shown in Figure 13.3. *Constrs* consists of an intersection of trace expressions of the form $FilterEventType_j \gg C_j^x$. It appears at the top level of the trace expression in an intersection ($\wedge$) with the repeating *Cyclic* step. This allows us to put constraints on belief events without considering at which action step they occur. In this way, each time a constrained belief event is observed in a *SingleStep*, we can keep track of the fact. $FilterEventType_j$ is an event type which denotes only the events involved in C_j^x. Its purpose is to filter out any events that are not constrained by C_j^x, and matches $bel(b_{j,1})$, $not_bel(b_{j,1})$, $bel(b_{j,2})$ and $not_bel(b_{j,2})$.

Each constraint represents a pairwise relationship between two belief events. These are captured by the three trace expressions in (14.9), (14.10) and (14.11) which describe the evolving behaviour of the four belief events of interest where $B_{j,i}$ is either the assertion or removal of $b_{j,i}$ and $NB_{j,i}$ is its converse – so

$$Constrs = \bigwedge_{j=1}^{o} FilterEventType_j \gg [C_j^1 \parallel C_j^2] \tag{13.8}$$

$$C_j^1 = (((B_{j,1}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon)) \cdot C_j^1) \vee (NB_{j,1} : C_j^2) \tag{13.9}$$

$$C_j^2 = (((NB_{j,1}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon)) \cdot C_j^2) \vee (B_{j,1}{:}C_j^1) \vee (B_{j,2}{:}C_j^3) \tag{13.10}$$

$$C_j^3 = (((B_{j,2}{:}\epsilon) \vee (NB_{j,1}{:}\epsilon)) \cdot C_j^3) \vee (NB_{j,2} : C_j^2) \tag{13.11}$$

$$B_{j,i} = [bel(belief_{j,i}) \parallel not_bel(belief_{j,i})] \tag{13.12}$$

$$NB_{j,i} = [bel(belief_{j,i}){:}\epsilon \parallel not_bel(belief_{j,i}){:}\epsilon] \tag{13.13}$$

FIGURE 13.3. Trace Expressions for Constrs where $B_{j,i} \neq NB_{j,i}$.

if $B_{j,i} = bel(b_{j,i})$ then $NB_{j,i} = not_bel(b_{j,i})$ and vice versa. The three equations capture the constraint that if $B_{j,1}$ has occurred then $B_{j,2}$ can not occur until after $NB_{j,1}$ has been observed and vice versa. The constraint either starts in the state described by C_j^1 or C_j^2 depending upon whether only one of the constrained belief events is possible in the initial state (C_j^1) or both are (C_j^2).

13.4.3 Abstract Model Generation

Once we have created a trace expression, we translate it into Java by implementing `add_random_beliefs`. We omit the involved low level details (e.g., constructing appropriate class and package names) but just focus on the core aspects[9]. Our trace expression is defined according to the template in Figures 13.2 and 13.3. Many parts of these trace expressions are not directly translated into Java; the sub-expressions relevant to the generation of abstract models are *Pre* (14.2), *SingleStep* (14.4) and *Constrs* (14.8). Note that the MCAPL framework provides support for constructing logical predicates and adding them to the belief base.

If *Pre* specifies particular initial beliefs then the subclass adds these to the agent's belief base at the start. *SingleStep* contains a union of trace expressions of the form $Action = action(action_name) : ProtocolBel$. $ProtocolBel = |_{i=1}^{k} (bel(b_i) \vee not_bel(b_i) \vee \epsilon$ defines the set of belief events that may occur. We define the set $\mathcal{B}(ProtocolBel)$ as $b_i \in \mathcal{B}(ProtocolBel)$ iff $(bel(b_i) \vee not_bel(b_i) \vee \epsilon)$ is one of the interleaved trace expressions in *ProtocolBel*.

For each $b_i \in \mathcal{B}(ProtocolBel)$ we define a predicate in the environment class and bind it to a Java field called b_i.

Constrs constrains events by specifying pairwise exclusions between some of them.

[9]Full source code can be found in the MCAPL distribution: `mcapl.sourceforge.net`. Code for the examples is also available from the University of Liverpool together with experimental data – DOI:`10.17638/datacat.liverpool.ac.uk/438`

For each *Action* trace expression we generate a corresponding *if statement* inside the add_random_beliefs method.

```
if(act.getFunctor().equals("action_name"))     1
     { translation(ProtocolBel, Constrs) }      2
```

We construct a set of mutually exclusive belief events, $\mathcal{M}_x(Constrs)$, from *Constrs* where $(B_{j,1}, B_{j,2}) \in \mathcal{M}_x(Constrs)$ iff $FilterEventType_j \gg Constraint_j$ is one of the conjuncts of *Constrs* and $C_j^1 = (((B_{j,1}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon)) \cdot C_j^1) \vee (NB_{j,1} : C_j^2)$ and $C_j^3 = (((B_{j,2}{:}\epsilon) \vee (NB_{j,1}{:}\epsilon)) \cdot C_j^3) \vee (NB_{j,2} : C_j^2)$.

The set of possible sets of belief events for our structured environment is:

$$\mathcal{PB}(ProtocolBel, Constrs) = \{S \mid (\forall b_i \in \mathcal{B}(ProtocolBel).\ bel(b_i) \in S \vee not_bel(b_i) \in S) \wedge (\forall (B_1, B_2) \in \mathcal{M}_x(Constrs).\ B_1 \in S \leftrightarrow B_2 \notin S)\} \quad (13.14)$$

Say that $\mathcal{PB}(ProtocolBel, Constrs)$ contains k sets of belief events, S_j, $0 \leq j < k$. We generate $translation(ProtocolBel, Constrs)$, as follows:

```
int assert_random_int = random_int_generator(k);     1
```

where random_int_generator is a special method that generates random integers in a way that optimises the model checking in AJPF. For each S_j we generate

```
if (assert_random_int == j)     1
  { add_percepts(S_j) }         2
```

Here $add_percepts(S_j)$ adds b_i to the percept base for each $bel(b_i) \in S_j$. We do not need to handle the belief removal events, $not_bel(b_i) \in S_j$, because AJPF automatically removes all percepts before calling add_random_beliefs.

Figures 13.4 and 13.5 show the trace expression modeling the cruise control agent from Example 7. *Pre* is reused for *AddBelEnv* since, in this case, they are the same trace expression. *SingleStep* contains only one branch which matches any action. *ProtocolBel* specifies that the possible belief events are the assertion and removal of *safe*, *at_speed_lim*, *driver_accelerates* and *driver_brakes*

We have two constraints. Firstly we assume that the driver never brakes and accelerates at the same time. This establishes a mutual exclusion between $bel(driver_accelerates)$ and $bel(driver_brakes)$. Initially either belief may appear. Secondly, we assume the driver only accelerates if it is safe to do so. This establishes a mutual exclusion between $bel(driver_accelerates)$ and $not_bel(safe)$.

Initially we are in the state were we cannot observe $bel(driver_accelerates)$. *brake_or_accelerate* and *accelerates_or_safe* are event types that match the relevant events for each constraint.

$$Protocol = Pre \cdot (Cyclic \wedge Constrs) \tag{13.15}$$
$$Pre = ((not_action{:}Pre) \vee \epsilon) \tag{13.16}$$
$$Cyclic = SingleStep \cdot Cyclic \tag{13.17}$$
$$SingleStep = action(any_action){:}(ProtocolBel \cdot Pre) \tag{13.18}$$
$$Safe = ((bel(safe){:}\epsilon) \vee (not_bel(safe){:}\epsilon) \vee \epsilon) \tag{13.19}$$
$$AtSpeedLimit = ((bel(at_speed_lim){:}\epsilon) \vee (not_bel(at_speed_lim){:}\epsilon) \vee \epsilon) \tag{13.20}$$
$$Accel = ((bel(driver_accelerates){:}\epsilon) \vee (not_bel(driver_accelerates){:}\epsilon) \vee \epsilon) \tag{13.21}$$
$$Brakes = ((bel(driver_brakes){:}\epsilon) \vee (not_bel(driver_brakes){:}\epsilon) \vee \epsilon) \tag{13.22}$$
$$ProtocolBel = (Safe|AtSpeedLimit|Accel|Brakes) \tag{13.23}$$

FIGURE 13.4. Trace expression for a Cruise Control Agent.

13.4.4 MCAPL Runtime Verification

Since the MCAPL framework is implemented in Java, its integration with the trace expressions runtime verification engine or "monitor" (namely, the Prolog engine that "executes" the δ transitions as presented in Section 14.2) was easy using the JPL interface[10] between Java and Prolog[11].

In order to verify a trace expression τ modelled in Prolog, we supply the runtime verification engine with Prolog representations of the events taking place in the environment. These are easily obtained from the abstraction engine and the Java environment that links to sensors and actuators. The Java environment reports instances of `assert_shared_belief`, `remove_shared_belief` and `executeAction` to the runtime verification engine which checks if the event is compliant with the current state of the modelled environment and reports any *violations* that occur during execution[12].

13.5 Discussion

AJPF's property specification language uses LTL extended with modalities for BDI concepts such as beliefs ($B(a, b)$ is interpreted as meaning agent a believes b)[13]

We carried out experiments using the agent discussed in Example 7. When model checked using a typical hand-constructed unstructured abstraction, verification takes 4,906 states and 32:17 minutes to verify that it is always the case that eventually the car believes is it safe or that it is in the process of

[10] `http://jpl7.org`
[11] We reuse pre-existing work used to develop RIVERtools and the JADE-Connector (Chapter 14).
[12] Following the approach presented in Section 14.2.
[13] See Section 3.6 for further details.

$$\begin{aligned}
Constrs &= (brake_or_accelerate \gg BrakeOrAccelerate) \wedge \\
&\quad (accelerates_or_safe \gg CanBeUnsafe) \qquad (13.24)\\
CanBeUnsafe &= (bel(safe){:}AccelOrUnsafe) \vee (((not_bel(safe){:}\epsilon) \vee \\
&\quad (not_bel(driver_accelerates){:}\epsilon)) \cdot CanBeUnsafe) \qquad (13.25)\\
AccelOrUnsafe &= (bel(driver_accelerates){:}CanAccel) \vee \\
&\quad (((not_bel(driver_acclerates){:}\epsilon) \vee (bel(safe){:}\epsilon)) \cdot \\
&\quad AccelOrUnsafe) \vee (not_bel(safe){:}CanBeUnsafe) \qquad (13.26)\\
CanAccel &= (not_bel(driver_accelerates){:}AccelOrUnsafe) \vee \\
&\quad (((bel(safe){:}\epsilon) \vee (bel(driver_accelerates){:}\epsilon)) \cdot CanAccel) \qquad (13.27)\\
BrakeOrAccelerate &= (bel(driver_accelerates){:}AccelOnly) \vee \\
&\quad (((not_bel(driver_accelerates){:}\epsilon) \vee \\
&\quad (not_bel(driver_brakes){:}\epsilon)) \cdot \vee \\
&\quad BrakeOrAccelerate)(bel(driver_brakes){:}BrakeOnly) \qquad (13.28)\\
AccelOnly &= (not_bel(driver_accelerates){:}BrakeOrAccelerate) \vee \\
&\quad (((bel(driver_accelerates){:}\epsilon) \vee (not_bel(driver_brakes){:}\epsilon)) \cdot \\
&\quad AccelOnly) \qquad (13.29)\\
BrakeOnly &= (not_bel(driver_brakes){:}BrakeOrAccelerate) \vee \\
&\quad (((bel(driver_brakes){:}\epsilon) \vee (not_bel(driver_accelerates){:}\epsilon)) \cdot \\
&\quad BrakeOnly) \qquad (13.30)
\end{aligned}$$

FIGURE 13.5. Trace Expression for the Constraints on a Car where the driver only accelerates when it is safe to do so, and never uses both brake and acceleration pedal at the same time.

braking:

$$\Box(B(car, safe) \rightarrow \Box(\Diamond B(car, safe) \vee B(car, braking))) \quad (P1)$$

The condition $B(car, safe) \rightarrow$ at the start of the formula considers the possibility that the car never believes it is safe since braking is only triggered when the *safe* belief is removed.

To test our approach, we first used the trace expression in Figure 13.4 with the omission of *Constrs*: this trace expression is equivalent to an unstructured abstraction, i.e., one where the percepts `safe`, `at_speed_lim`, `driver_brakes`, and `driver_accelerates` could all either be true or false at any moment. Verifying (P1) in an abstract model generated from this trace expression took 4,906 states and 30:37 minutes: the behaviour was exactly the same as that for the unstructured model that had been created manually, and this helped validate that trace expressions following the template in Figure 13.2 without constraints create unstructured abstractions that behave the same way as hand crafted

ones.

We then investigated the effect of structuring the model using the trace expression in Figure 13.5, which adds constraints to that in Figure 13.4. With this abstraction (P1) takes 8:22 minutes to prove using 1,677 states – this has more than halved the time and the state space.

To illustrate how we cope with the risk that a structured abstraction may not reflect reality, we consider a version of the cruise control agent with slight variations. It is widely considered important that an autonomous vehicle *should not* be able to override the actions of a driver. In our previous example the vehicle violates this rule – it would only let the driver accelerate if it was safe to do so, and it would brake *whenever* it detected unsafe conditions even if the driver was currently trying to accelerate. We adapted the program, removing these restrictions.

This modified program could *not* be verified in the unstructured model because our property is *not* actually true in that model – if the driver continually accelerates in an unsafe situation then the car can *never* brake. However, it is true in the structured model which assumes that the driver never accelerates if the situation is unsafe.

When we run this program in our simulator it is indeed possible to cause a crash by accelerating in unsafe conditions. This is where the runtime verification engine fits in. The engine logs an exception at the moment when the unsafe acceleration takes place. It generates the error message shown below and also shows the current state of the trace expression, which is the equivalent of (14.25) in Figure 13.5.

```
*** DYNAMIC TYPE—CHECKING ERROR ***
Message event(abstraction_car0,
            assert_shared(driver_accelerates))
cannot be accepted in the current state

S_8=(bel(safe):S_6)\/((not_bel(safe):epsilon)\/
(not_bel(driver_accelerates):epsilon))*S_8])
```

This identifies the system as now being in an unverified state, as this acceleration has violated the trace expression.

The example shows how we have addressed the development of a principled mechanism for creating structured abstractions in a way that allows us to provide at least some guarantee of the validity of our results. We have demonstrated how trace expressions can be used as a unifying formalism to generate both a structured abstraction for model checking and a runtime monitor, providing a route for guarantees of the behaviour of a system that has been verified against an abstract model of the real world. Their expressive power would pave the way to addressing challenging scenarios where:

1. the behaviour of the system is modeled with a trace expression τ without expressive power limitations (for example, an expression representing the set of all $a^n b^n$ traces, for any $n \in \mathbb{N}$; this set of traces cannot be

modeled in LTL) to allow specifications of complex environments;

2. τ is over-approximated (as shown in Chapter 12) and then translated into the Java model;

3. the model checking stage is performed using the generated over-approximating Java model;

4. the runtime verification stage uses τ, with all its expressive power; empirical results show that in most cases verifying whether a trace belongs to the language defined by a trace expression is linear in the lenght of the trace: this means that – even when the highest modeling expressivity of the formalism is exploited – performances of RV remain acceptable.

Part VI

Implementation and Case Study

In this part we present the tool that has been developed to support the trace expression RV. Thanks to an IDE especially developed for supporting the trace expression formalism, we can achieve the writing of properties and protocols using trace expressions more intuitive, where the syntax and event types are statically checked and the monitors are automatically generated.

14 *Development of a framework supporting trace expressions RV*

"We are stuck with technology when
what we really want is just stuff that works."
- Douglas Adams

The development of new formalisms and libraries to solve well known problems is common in all computer science fields. What is less common is to find tools that make such new formalisms actually usable also by those users which were not directly involved in the development of the formalism itself. The same occurs in the context of Runtime Verification, where many formalisms and libraries have been developed during the years, but only few of them are supported by an Integrated Development Environment (IDE) that makes them accessible to a wider audience. This chapter introduces RIVERtools, the IDE which has been developed to support the definition of trace expressions and their use for obtaining the runtime verification of our systems.

The contents of this chapter are published in
(Ancona et al., 2018b; Ferrando, 2017)

14.1 Introduction

Differently from static verification, very few tools expressly designed for the runtime verification of MAS exist. Besides (Alotaibi and Zedan, 2010; Bakar and Selamat, 2013; Chesani et al., 2009; Meron and Mermet, 2006) and a few others, the only works on runtime verification of MAS are those that lead to the development and refinement of the trace expression formalism[1].

As it happened for tools such as Jason (Section 3.3), Cartago[2], and Moise[3], where the need of a more modular, flexible and standardized framework brought to the creation of JaCaMo[4], also for the RV of MAS there is the pressing need of a general purpose framework, with the objective to be modular with respect to the target system (the system must be seen as a black-box) and "programmer-friendly" focusing on the reuse of the developed software.

Following this aim we developed RIVERtools, an Integrated Development Environment (IDE) for supporting the automatic generation of code to be used for implementing a black-box RV engine for generic software systems - focusing on challenging scenarios such as MAS.

RIVERtools is a *generic*, *modular* and *domain independent* IDE for achieving the runtime verification of any possible target system. Rather than other solutions proposed in literature, RIVERtools was born as a "multi-target" IDE for the generation of runtime monitors.

Obviously we can achieve the same results without using an IDE, but using RIVERtools we have the support during the definition of our properties (supported by highlights, syntax and type checking), leaving the technical details at a lower level that can be implemented just once for each possible target system of interest (updating the compiler without changing the high level specification).

The RIVERtools main features can be summarized as follows:

- the support during the definition of our properties (using the trace expression formalism);
- the abstraction from the specific target domain, with RIVERtools we can define the specification leaving all technical details to the compiler (presented in Section 14.3);
- the automatic properties integration inside the runtime verification process;
- the extensibility because the RIVERtools' structure is based on the presence of "connectors-to-something" (for each new target system to be verified, we define a new "connector-to-target" updating the RIVERtools compiler).

[1] `https://www.google.it/search?q=runtime+verification+of+multiagent+systems`
[2] `http://cartago.sourceforge.net`
[3] `http://moise.sourceforge.net`
[4] `http://jacamo.sourceforge.net`

The main objective of RIVERtools is to separate and generalize the writing of monitors to achieve the runtime verification of any possible target system.

In this chapter we focus on its initial exploitation for the MAS scenario, in particular for the JADE framework (Section 3.4).

14.2 Trace expression RV using SWI-Prolog

As it has been repeated several times in this thesis, we have totally implemented the trace expression syntax and semantics using SWI-Prolog.

The main reasons for this choice can be found in the support for defining cyclic terms natively offered by SWI-Prolog. Thanks to this simplification, the translation of a trace expression in its counterpart implemented in SWI-Prolog is extremely direct. Also the trace expressions operational semantics can be easily represented inside SWI-Prolog, translating each rule in an equivalent logic predicate. Finally, not for importance, SWI-Prolog is a well-know framework and is one of the most used. Thanks to this, the state of the art is full of integration libraries that allow interacting with other programming languages; in our case, since the target system will be implemented using JADE, we will exploit the integration with the Java language.

Trace expressions can be modelled as Prolog terms and by exploiting syntactic equations where the same variable can appear both to the left and to the right of the "=" syntactic equality symbol, recursive trace expressions can be easily defined (points 1 and 2 in the list below). This feature is supported by most Prolog implementations, including SWI-Prolog, and allows us to define the trace expressions shown in the examples provided so far, using almost the same syntax. The adoption of Prolog is a winning choice not only for representing trace expressions, but also for implementing their semantics and for manipulating them.

Provided that some connector between the real system to be observed and a SWI-Prolog piece of code exists or can be implemented (point 3 in the list below), the development of a SWI-Prolog trace expression-driven monitor observing the events and verifying whether they comply with the trace expression or not, can be automatically generated from the trace expression Prolog representation (point 4 and 5 in the list below). This set of events can be generated by any possible system (black-box approach), the only need is that it must be able to communicate with our SWI-Prolog specification.

The states are maintained by SWI-Prolog in its local knowledge base, to allow the monitor to retrieve the current set of states, query each of them, and update the knowledge base with newly generated states.

Having in mind these considerations, the corresponding RV pipeline can be summarized as:

1. the implementation of the operational semantics modeled by δ (Section 4.2.3) in SWI-Prolog;

2. the definition of a trace expression in SWI-Prolog;

3. the implementation of a library to allow the communication between SWI-Prolog and the target system we want to verify (in the following we will refer to it as the connector);

4. the writing of a "main" file which uses this library;

5. the execution of the "main" file to perform the runtime verification of the target system.

What is extremely important to note is that the components involved in the RV process are: a SWI-Prolog library (1), a trace expression (2), a library to integrate SWI-Prolog with the system (the connector) (3) and a file to start everything correctly (4). But among these, only the trace expression (2) needs to be defined each time by the programmer (inside SWI-Prolog as introduced above). In fact, when the target system is chosen, (1) and (3) can be implemented once and for all; and the "main" file (4) can be automatically generated starting from the trace expression (2).

Following this intuition, we created an IDE which supports the definition of our specifications through trace expression and that automatically generates of all the code necessary to achieve the runtime verification of our target system. As we explained before, RIVERtools is based on the concept of "connector-to-target"; in the following, we consider its exploitation using the "connector-to-JADE" in order to verify our MAS implemented using JADE[5].

14.3 RIVERtools

RIVERtools has been developed using *Xtext*[6], a framework for the development of programming languages and domain-specific languages which can be integrated as an Eclipse[7] plugin. With respect to other frameworks, Xtext has been chosen above all for its support to the *Xtend*[8] language (a dialect of Java), that makes the development of Eclipse plugin very fast and intuitive.

In Figure 14.1(left), a high level summarization of how the RIVERtools IDE works is presented. First of all, the programmer (the user) can write the trace expression inside RIVERtools. This phase is supported by:

- syntax checking,
- type checking (contractiveness, decentralizable, and so on),
- roles, event types and events definition, and so on.

Referring to the second point, a trace expression is contractive if all its infinite paths contain the *prefix* ':' operator (Section 4.2.3) and can be decentralized (for decentralized runtime verification purposes) only if it satisfies a set of good

[5]JADE is the only target system supported by RIVERtools for now.
[6]`https://eclipse.org/Xtext/`
[7]`https://www.eclipse.org`
[8]`http://www.eclipse.org/xtend/`

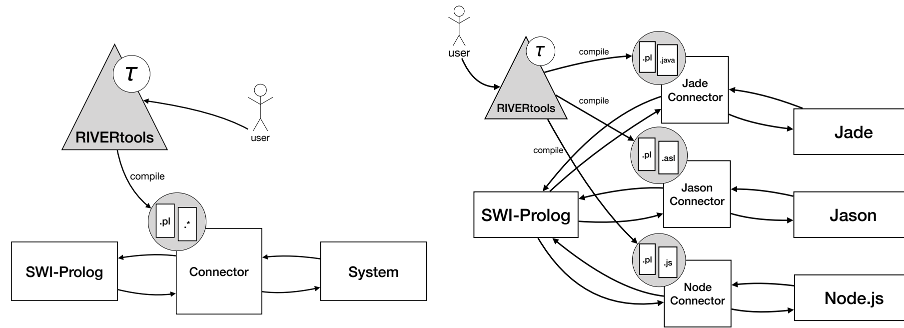

FIGURE 14.1. RIVERtools general representation (left) and its exploitation in three different scenarios: JADE, Jason and Node.js (right).

properties: *connectedness for sequence* and *unique point of choice* (Chapter 8 and Chapter 9).

Both contractivennes and decentralizability checks are provided by RIVER-tools.

Keeping in mind the summarization made in Section 14.2, we have a trace expression defined using RIVERtools, the SWI-Prolog library and the connector to the target system.

Starting from these, we need to create the SWI-Prolog representation of the trace expression which will be used inside the SWI-Prolog library (where the trace expression operational semantics is implemented), and the "main" file to properly initialize and run the connector. These two files are automatically generated by RIVERtools by compiling the trace expression.

In more detail, the new RV pipeline using the trace expression formalism with the RIVERtools integration can be summarized as:

1. the user writes the trace expression inside RIVERtools and chooses the target system;

2. RIVERtools checks the correctness of the trace expression and, if it is correct, generates the SWI-Prolog representation of the trace expression and the "main" file for the target system;

3. the "main" file, using the SWI-Prolog library and the connector to the target system, can be used by the user to start the RV process.

Figure 14.1(right) shows the flexibility of the proposed approach. As an example, if the target system is JADE, RIVERtools compiles the trace expression τ into one SWI-Prolog file and one Java file: the second file is indeed dependent on the target system; for JADE it is a Java file, for Jason it would be an ASL file and for Node.js it would be a Javascript one.

The grammar

The trace expression structure that is recognized and managed by RIVERtools is expressed by the grammar below.

```
⟨ trace_expression ⟩ ::= 'trace_expression' '{'
    'id:' ⟨ atom ⟩
    'target:' ⟨ target ⟩
    'body:' ⟨ term ⟩+
    'roles:' ⟨ role ⟩*
    'types:'
       (⟨ type ⟩':' '{'
           (⟨ role ⟩ '=>' ⟨ role ⟩ ':' ⟨ content ⟩ ('[' (⟨ condition ⟩)+ ']')?)+
       }' '[' ⟨ channel ⟩ ']')*
    ('threshold:' ⟨ reliability ⟩)?
    ('channels:' (⟨ channel ⟩ ('[' ⟨ reliability ⟩ ']')?)+)?
'}'
```

There are seven different fields:

- *id:* is the identifier of the protocol (the protocol name for instance);
- *target:* is the target system to verify (only JADE for now);
- *body:* is the collection of terms representing the body of the protocol, it follows the trace expression syntax presented in Section 4.2.3;
- *roles:* is the set of roles involved in the event types of the protocol (the roles are the agents since the target is set to JADE);
- *types:* is the set of event types used in body;
- *threshold:* (optional) is the minimum level for the reliability allowed by the protocol (it is the likelihood for the channel, see Chapter 8);
- *channels:* (optional) is the set of channels available for the communication between the roles.

We can better understand the syntax through an example.

```
interaction_trace_expression {
  id: ping_pong
  target: jade
  body:
    main <- ping : pong : main
  roles:
    alice
    bob
  types:
    ping : { alice => bob : hello } [email]
    pong : { bob => alice : world } [sms]
  threshold: 0.7
  channels:
    email [0.8]
    sms
}
```

This is a simplified representation of the ping-pong protocol presented in Section 4.3.1.

Since the target system is set to `jade` – the only available for now – the roles involved are automatically considered as agents, in particular here we have two agents involved: `alice` and `bob`.

Since the `target` field is set to `jade`, the roles are interpreted as the name of the agents involved in the interaction protocol. For this reason, in the entire chapter we will consider the terms role and agent synonyms.

The protocol identifier is `ping_pong`, and its definition (the body) consists in one `ping` followed by one `pong` followed by one `ping` and so on infinitely.

The event types involved in the protocol are explicitly defined:

- `ping` corresponds to the interaction event `hello` sent by `alice` to `bob` using the `email` channel, and
- `pong` corresponds to the interaction event `world` sent by `bob` to `alice` using the `sms` channel.

These two channels must be also defined with their respective reliabilities (likelihoods, from `0`, not reliable, to `1`, totally reliable); for instance in this case we have that the `sms` channel (the deault is `1` if not specified otherwise) is more reliable than the `email` channel (`0.8`).

In RIVERtools the concept of reliability of the channels is the implementation corresponding to the likelihood concept presented in Chapter 8.

The threshold field is used to set the level of reliability under which we do not trust a channel. When the reliability of a channel is under the threshold RIVERtools automatically considers "optional" all the event types using that channel. For instance, if we have a channel `sms` with reliability `0.3`, the threshold set to `0.7`, and an event type `ping:{alice => bob : hello}[sms]`; each time we use `ping` inside the body field, *i.e.* `ping:t` (where `t` is the continuation of the trace expression after the prefix operator), RIVERtools automatically compiles it in `((ping:epsilon)\/(epsilon))*(t)` meaning that the event matching `ping` can be missing (`*` is the corresponding representation of the concatenation operator $\cdot$ inside RIVERtools, the syntax for the other operators is unchanged). If we set the threshold to a smaller value, for instance `0.2`, we are relaxing our reliability requirements and, since the `sms` channel has a greater reliability value, it will not be modified by RIVERtools because we are trusting all the channels with reliability greater than `0.2`.

In the case of a limit scenario where the reliability of a channel is `0`, RIVERtools will remove from the protocol all the occurrences of the event types using it. This particular case becomes interesting when we focus on distribution features, because we can write a good protocol in which we explicit all the event interactions but when we exploit it in a real world we discover that not all events are observable (partial observability problem, as we already presented in Section 8.5). This issue can bring to have missing information during the decentralization of our specifications. Thanks to the explicit representation of such reliability absence, RIVERtools can analyze the trace expression considering these kind of unavailability and can achieve a correct decentralization without losing information (exploiting the Decamon algorithm, see Chapter

9).

The channel integration inside the runtime verification process is a task for the connector and it is totally domain dependent.

When a channel is not indicated for an event type, the default one is chosen whose reliability (likelihood) is `1`. The same happens for the channels, if the reliability is not given, the reliability is set to `1` (in the `ping_pong` example the `sms` channel has reliability `1`).

Starting from the trace expression, RIVERtools generates its SWI-Prolog implementation and the system dependent "main" file (in this case a Java file since the target is JADE) to use the JADE-Connector and the SWI-Prolog library. The programmer will provide its own MAS implemented in JADE and using the "main" file it will be possible to perform RV on it.

In more details, since JADE is a Java framework used to implement MAS, the JADE-Connector is a Java library containing both all the primitives to sniff the events generated by the agents developed by the programmer and the code necessary to use the SWI-Prolog library containing the trace expression semantics implementation. In this way, the only part that RIVERtools must generate is the SWI-Prolog implementation of the trace expression, all the rest is fixed and can be reused (Figure 14.2). It will be enough to run the "main" file (Java) providing:

- the agents developed by the programmer;
- the JADE-Connector library (in the build path);
- the SWI-Prolog library (used by the JADE-Connector).

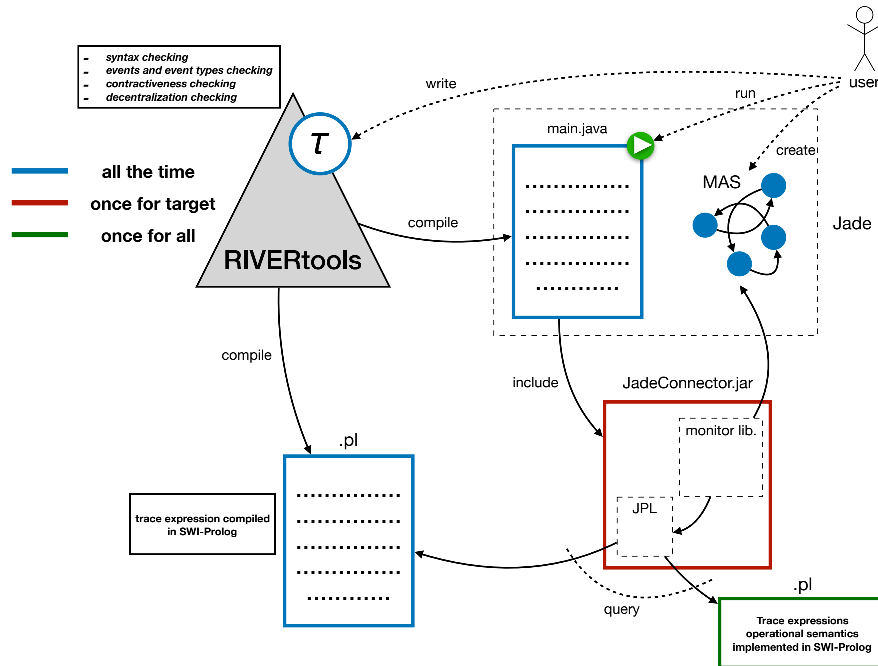

Figure 14.2. RIVERtools exploited in JADE.

In Figure 14.2 we used colours to differentiate the framework parts taking into account the timing. Since the trace expression represents the protocol we want to verify at runtime, it is the part that changes in each new scenario. Consequently, also the automatically generated SWI-Prolog implementation and "main" file change each time (all these parts are in blue). The target system is JADE, thus we have a JADE-Connector that implements all the monitoring process logic and the communication with the SWI-Prolog library. The connector changes only when we change the target system, since here we are focusing on JADE, the connector is used for all the possible scenarios implemented in JADE (the red part). Trace expressions have been successfully implemented in SWI-Prolog, with their syntax and semantics. Since one of the objectives of the connector is the communication with the SWI-Prolog library, the implementation of the trace expression formalism has been done once and for all. If in the future we will be interested in adding a new target system, it will be enough developing the corresponding connector in order to communicate with the SWI-Prolog implementation.

Before showing an example directly implemented using RIVERtools, we need to clarify better what is the JADE-Connector and the SWI-Prolog library.

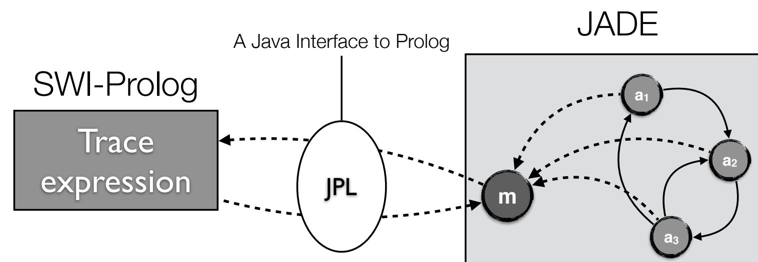

FIGURE 14.3. Abstract view of how JPL is used inside the JADE-Connector.

The JADE-Connector is the Java library that has been developed during the Ph.D. program in order to monitor MAS implemented in JADE. In order to use this library is enough to add it to the build path of a JADE project. After that, we can use the library to create and run one or more monitors simply calling the corresponding implemented functions.

The SWI-Prolog library (on the left in Figure 14.3, and in the right bottom of Figure 14.2) handles all the issues concerning the representation of our trace expressions. In order to use this library inside the JADE-Connector, we exploited JPL[9] (Java interface to ProLog). Using JPL (Figure 14.3 and Figure 14.2) is possible to call SWI-Prolog predicates directly from Java. In this way, with the JADE-Connector we are able to observe the events generated inside our MAS in JADE, and through JPL, we can query our trace expression represented and maintained inside SWI-Prolog[10]. As previously observed, and graphically

[9] `http://www.swi-prolog.org/packages/jpl/java_api/index.html`

[10] The JADE-Connector, the SWI-Prolog JPL integration and the RIVERtools eclipse plugin projects are all available on GitHub: `https://github.com/AngeloFerrando/`.

showed in Figure 14.2, this part is totally independent from the current system and specification used. Thanks to this, we can reuse our JADE-Connector for every possible scenarios in JADE. It is enough to change: the target MAS and the protocol.

14.3.1 An example using RIVERtools

Reconsidering the book shop example presented in Section 8.4. Let us suppose to have a MAS implemented in JADE representing a it. The involved agents are: `alice`, `barbara`, `carol`, `dave`, `emily` and `frank`. For clarity we briefly reformulate in natural language the book shop example as follow:

1. the agent `alice` sends a `whatsApp` message to the agent `barbara` asking to buy a book;

2. the agent `barbara` sends an `email` message to the agent `carol` asking to reserve the book in the bookshop;

3. the agent `carol` sends a `whatsApp` message to the agent `dave` asking to check the availability of the book;

4. the agent `dave` checks the availability of the book, and if the book is available

   a) it sends a `whatsApp` message to the agent `emily` asking to take the book in the bookshop;
   b) the agent `emily` sends an `email` message to the agent `barbara` saying that the book is available in the bookshop.

   otherwise

   a) it sends an `email` message to the agent `frank` asking to order the book;
   b) the agent `frank` sends a `whatsApp` message to the agent `barbara` saying that the book will be availabel in the bookshop in two days.

The corresponding trace expression representation inside RIVERtools is:

Listing 14.1. Book-shop trace expression written inside RIVERtools.

```
interaction_trace_expression {
  id: book_purchase
  target: jade
  body:
    main <- buy : reserve : checkAvail :
                 (take2Shop : availNow : epsilon \/
                  order : avail2Days : epsilon)
  roles:
    alice, barbara, carol, dave, emily, frank
  types:
    buy : { alice => barbara : buy_me_book} [whatsApp]
    reserve : { barbara => carol : reserve_me_book } [email]
    checkAvail : { carol => dave : is_available? } [whatsApp]
    take2Shop : { dave => emily : send_me_book } [whatsApp]
    availNow : { emily => barbara : book_available } [email]
    order : { dave => frank : order_book } [email]
    avail2Days : { frank => barbara : book_in_2_days } [whatsApp]
  threshold: 0.6
  channels:
    email [0]
    whatsApp [1]
}
```

In this example we have two kinds of channels used by the agents to communicate: `email` and `whatsApp`. We point out the reliability of the two corresponding channels. The `whatsApp` channel has reliability set to `1`, while the `email` channel has the reliability set to `0`. This means that, during RV, we expect the monitor not to be able to observe messages passed on the `email` channel. This may be due to any reason, for example lack of permissions or unavailability of information. Thanks to the presence of explicitly declared channels, RIVERtools can take into account the channel reliability during the correctness checking for the trace expression. Channel reliability may impact on contractiveness and decentralization of the original trace expression (as observed in Section 14.3 and Chapter 8): RIVERtools takes channel reliability into account when automatically performs those checks.

Compiling this trace expression, RIVERtools generates the two files

- `book_purchase.pl` and
- `BookPurchase.java`.

Providing the connector library for JADE and the MAS implementation of the book-shop, it will be enough to execute the main method of the `BookPurchase` class to achieve the RV of a JADE MAS. Once obtained this Java class and the trace expression SWI-Prolog representation, the developer can simply execute the verifier without adding a single line of code.

Listing 14.2. BookPurchase.java automatically generated by RIVERtools.

```
public class BookPurchase {
  public static void main(String[] args)
    throws StaleProxyException, IOException {

  /* Call at the SWI-Prolog library */
  JPLInitializer.init();

  /* Registration of the trace expression generated by RIVERtools */
  TraceExpression tExp = new TraceExpression("book_purchase.pl");

  /* Initialize JADE environment */
  jade.core.Runtime runtime = jade.core.Runtime.instance();
  Profile profile = new ProfileImpl();
  AgentContainer container = runtime.createMainContainer(profile);

  /* Create the custom defined agents
     (default are instances of Agent class) */
  List<AgentController> agents = new ArrayList<>();
  Agent alice = new Agent();
  AgentController aliceC = container.acceptNewAgent("alice", alice);
  agents.add(aliceC);

  // the same for barbara, carol, dave, emily and frank
  // ...

  /* Create a single centralized monitor
     verifying the trace expression tExp */
  SnifferMonitorFactory
    .createAndRunCentralizedMonitor(tExp, container, agents);

  /* Channels creation (here are simulated)*/
  Channel.addChannel(new SimulatedChannel("email", 0));
  Channel.addChannel(new SimulatedChannel("whatsapp", 1));

  /* Run the agents */
  for(Agent agent : agents) {
    agent.start();
  }
}
```

The Java class reported in Listing 14.2 is what we have been defining for the entire chapter the "main" file. The BookPurchase Java class is actually the content of the "main" file and is automatically generated starting from the trace expression defined inside RIVERtools. Looking at the code, we can see how this class makes full use of the JADE-Connector, starting from the JPLInitializer class, used for initializing the SWI-Prolog environment. Inside the JADE-Connector naturally we find also the corresponding implementation of trace expressions (TraceExpression Java class). And, thanks to the presence of a SnifferMonitorFactory, we can create and run the monitors (sniffers) very easily.

As already mentioned before, the JADE-Connector can be used directly as a Java library, it is enough to include it into the java build path. The class

automatically generated and reported in Listing 14.2 could have been written manually without using RIVERtools. But, using RIVERtools we have all the advantages of syntax and type checking on trace expressions, without the necessity to write Java code that can be automatically generated and that is more or less always the same[11].

14.3.2 Decentralizing the Example with RIVERtools

In Chapter 8 and Chapter 9 we presented and solved the problem of how to decentralize our global AIPs represented using trace expressions. These features are also included inside RIVERtools.

Considering again the book-shop example, inside RIVERtools we can define more complex aspects related to the verification of properties, as it follows:

Listing 14.3. Manual decentralization of book-shop trace expression in RIVERtools.

```
interaction_trace_expression {
  id: book_purchase
  target: jade
  body:
    main <- buy : reserve : checkAvail :
                 (take2Shop : availNow : epsilon \/
                  order : avail2Days : epsilon)
  roles:
    alice , barbara , carol , dave , emily , frank
  types:
    buy : { alice => barbara : buy_me_book} [whatsApp]
    reserve : { barbara => carol : reserve_me_book } [email]
    checkAvail : { carol => dave : is_available? } [whatsApp]
    take2Shop : { dave => emily : send_me_book } [whatsApp]
    availNow : { emily => barbara : book_available } [email]
    order : { dave => frank : order_book } [email]
    avail2Days : { frank => barbara : book_in_2_days } [whatsApp]
  threshold: 0.6
  channels:
    email [0]
    whatsApp [1]
  decentralized: true
  partition: [[alice] [barbara dave] [emily carol] [frank]]
}
```

By setting `decentralized` to true, we are saying to RIVERtools that we want to decentralize the RV of the MAS. We can also suggest the partition of agents that will be used for guiding the decentralization of the monitors at runtime. In this specific scenario we decided to monitor `barbara` and `dave` together, `emily` and `carol` together, and `alice` and `frank` separately. In this way we will generate 4 different monitors at runtime for the 4 different parts composing our chosen partition.

[11] Something that would easily end in repetitive copy and paste of Java code.

Listing 14.4. BookPurchase.java automatically generated by RIVERtools where we decentralize the RV on a fixed partition.

```
public class BookPurchase {
  public static void main(String[] args)
    throws StaleProxyException, ..., DecentralizedPartitionNotFoundException {

  /* Call at the SWI-Prolog library */
  JPLInitializer.init();

  /* Registration of the trace expression generated by RIVERtools */
  TraceExpression tExp = new TraceExpression("book_purchase.pl");

  /* Initialize JADE environment */
  jade.core.Runtime runtime = jade.core.Runtime.instance();
  Profile profile = new ProfileImpl();
  AgentContainer container = runtime.createMainContainer(profile);

  /* Create the custom defined agents
     (default are instances of Agent class) */
  List<AgentController> agents = new ArrayList<>();
  Agent alice = new Agent();
  AgentController aliceC = container.acceptNewAgent("alice", alice);
  agents.add(aliceC);

  // the same for barbara, carol, dave, emily and frank
  // ...

  /* Create and Set the partition */
  List<List<String>> groups = new ArrayList<>();
  List<String> group = new ArrayList<>();   groups.add(group);
  group.add("alice"); /* [alice] */
  group = new ArrayList<>(); groups.add(group);
  group.add("barbara"); group.add("dave"); /* [barbara dave] */
  group = new ArrayList<>(); groups.add(group);
  group.add("emily"); group.add("carol"); /* [emily carol] */
  group = new ArrayList<>(); groups.add(group);
  group.add("frank"); /* [frank] */
  Partition<String> partition = new Partition<>(groups);

  /* Decentralized monitors */
  for(Monitor m :
        SnifferMonitorFactory
          .createDecentralizedMonitors(tExp, partition, agents)){
    container.acceptNewAgent(m.getMonitorName(), m).start();
  }

  /* Channels creation (here are simulated)*/
  Channel.addChannel(new SimulatedChannel("email", 0));
  Channel.addChannel(new SimulatedChannel("whatsapp", 1));

  /* Run the agents */
  for(Agent agent : agents) {
    agent.start();
  }
}
```

RIVERtools does not only support the suggestion of a partition, but also integrates the DecAMon algorithm, presented in Chapter 9. Thanks to the DecAMon algorithm, RIVERtools can analyze the trace expression with respect to the proposed partition, and can understand if the partition is a monitoring safe partition or not (Definition 11). If the suggested partition is not monitoring safe, RIVERtools will raise an error asking the developer to modify it.

RIVERtools supports also a more automatic way to decentralize the specification without asking the developer to suggest a specific one. To be more clear, RIVERtools supports the automatic generation of monitoring safe partitions. If the developer wants to decentralize the runtime verification without caring about which (monitoring safe) partition has been selected, RIVERtools allows such kind of specification.

Modifying again the same example, we can have as follows.

Listing 14.5. Automatic decentralization of book-shop trace expression in RIVERtools.

```
interaction_trace_expression {
  id: book_purchase
  target: jade
  body:
    main <- buy : reserve : checkAvail :
                 (take2Shop : availNow : epsilon \/
                  order : avail2Days : epsilon)
  roles:
    alice , barbara , carol , dave , emily , frank
  types:
    buy : { alice => barbara : buy_me_book} [whatsApp]
    reserve : { barbara => carol : reserve_me_book } [email]
    checkAvail : { carol => dave : is_available? } [whatsApp]
    take2Shop : { dave => emily : send_me_book } [whatsApp]
    availNow : { emily => barbara : book_available } [email]
    order : { dave => frank : order_book } [email]
    avail2Days : { frank => barbara : book_in_2_days } [whatsApp]
  threshold: 0.6
  channels:
    email [0]
    whatsApp [1]
  decentralized: true
  minimal: true
  constraints:
    alice -><- carol
    barbara <--> dave
  number_of_monitors: [2,4]
  roles_for_monitor: [3,5]
}
```

This time we introduced 4 different new fields inside the specification. The `minimal` field is used by the developer in order to inform RIVERtools that the partition that will be automatically generated and used for RV must be a

minimal monitoring safe partition (Definition 8). The other fields allow limiting the structure of the generated partition. In particular:

- `constraints` contains all the constraints on the agents involved inside the partition, we can force two agents to be monitored together (`alice -><- carol`), and we can also force two agents to be monitored separately (`barbara <-> dave`).
- `number_of_monitors` sets the minimum and the maximum number of monitors we want to use for the runtime verification process (in this case we are saying to RIVERtools that we want at least `2` and at most `4` monitors).
- `roles_for_monitor` sets the minimum and the maximum number of roles monitored by a single monitor (in this case we are saying to RIVERtools that we want that each monitor verifies at leat `3` roles and at most `5` roles).

Thanks to these three fields we can customize the generation of the partition that will be used for obtaining the runtime verification of our MAS (Listing 14.6). These constraints depend on the specific scenario where RIVERtools is used.

Listing 14.6. BookPurchase.java automatically generated by RIVERtools where we decentralize the RV on a not specified minimal monitoring safe partition.

```
public class BookPurchase {
  public static void main(String[] args)
    throws StaleProxyException, ..., DecentralizedPartitionNotFoundException {

  /* Call at the SWI-Prolog library */
  JPLInitializer.init();

  /* Registration of the trace expression generated by RIVERtools */
  TraceExpression tExp = new TraceExpression("book_purchase.pl");

  /* Initialize JADE environment */
  jade.core.Runtime runtime = jade.core.Runtime.instance();
  Profile profile = new ProfileImpl();
  AgentContainer container = runtime.createMainContainer(profile);

  /* Create the custom defined agents
     (default are instances of Agent class) */
  List<AgentController> agents = new ArrayList<>();
  Agent alice = new Agent();
  AgentController aliceC = container.acceptNewAgent("alice", alice);
  agents.add(aliceC);

  // the same for barbara, carol, dave, emily and frank
  // ...

  /* Create and Set the partition */
  List<Condition<String>> constraints = new ArrayList<>();
  constraints
    .add(ConditionsFactory.createMustBeTogetherCondition("alice","carol"));
  constraints
    .add(ConditionsFactory.createMustBeSplitCondition("barbara","dave"));
  constraints
    .add(ConditionsFactory.createNumberOfConstraintsCondition(2,4));
  constraints
    .add(ConditionsFactory.createNumberAgentsForConstraintCondition(2,5));

  /* Get the first monitoring safe partition available */
  Partition<String> partition =
    tExp.getFirstMonitoringSafePartition(constraints);
  /* If no monitoring safe partition satisfying the constraints is */
  /* available, an exception is thrown! */

  /* Decentralized monitors */
  for(Monitor m :
        SnifferMonitorFactory
          .createDecentralizedMonitors(tExp, partition, agents)){
    container.acceptNewAgent(m.getMonitorName(), m).start();
  }

  /* Channels creation (here are simulated)*/
  Channel.addChannel(new SimulatedChannel("email", 0));
  Channel.addChannel(new SimulatedChannel("whatsapp", 1));

  /* Run the agents */
  for(Agent agent : agents) {
    agent.start();
  }
}
```

14.3.3 Screenshots

In this section we report screenshots showing some of the typical errors handled by RIVERtools.

THE BOOK-SHOP SCENARIO IN RIVERTOOLS: In Figure 14.4 we have defined inside RIVERtools the book-shop trace expression presented in Section 14.3.1. In this particular example, we have decided to decentralize the RV on a specific partition (the most distributed one where all the agents are monitored separately).

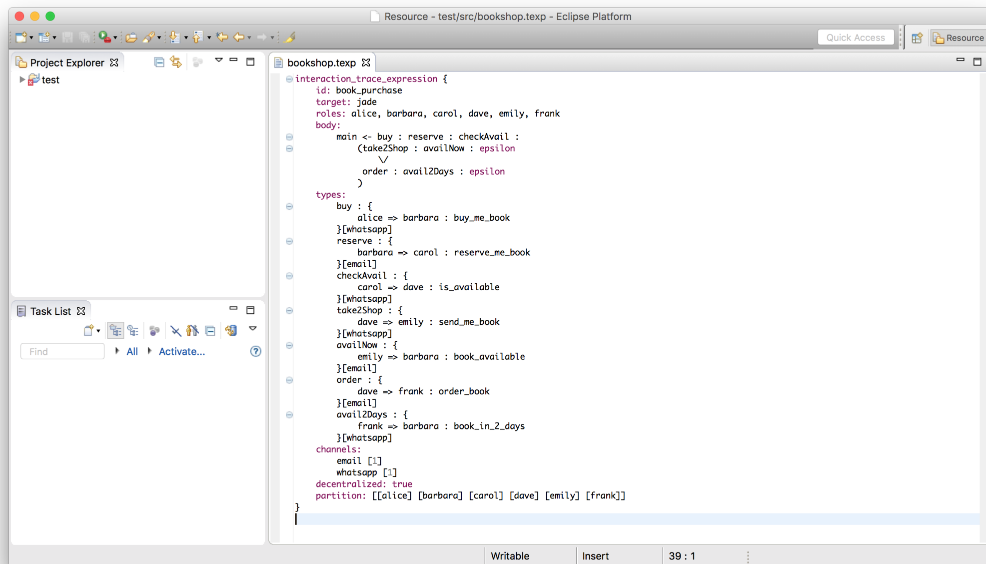

FIGURE 14.4. The book-shop scenario presented in Section 14.3.1.

PARTITION NOT VALID: In Figure 14.5 we change the reliability of the `email` channel from `1` to `0`. Unfortunately, this does not allow us to distribute the runtime verification on each single role because we could lose information. This is handled by RIVERtools that communicates to the programmer that the current proposed partition is not valid (because two critical points are not satisfied, see Definition 7).

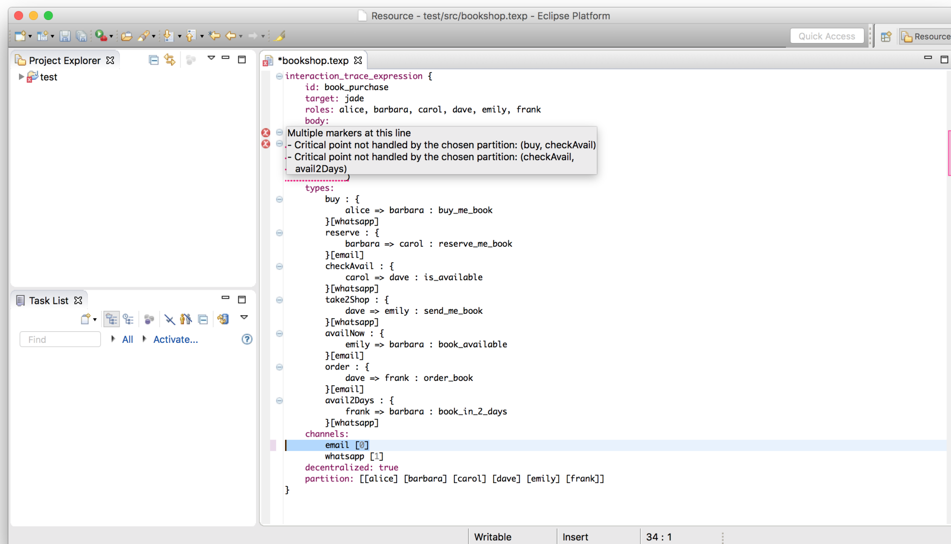

Figure 14.5. Error: Partition not valid

roles existence: In Figure 14.6 we remove a role from the `roles` set. Since this role is used inside the definition of two event types, RIVERtools informs the programmer about an existence problem, since it is not able to find the corresponding role inside the `roles` set.

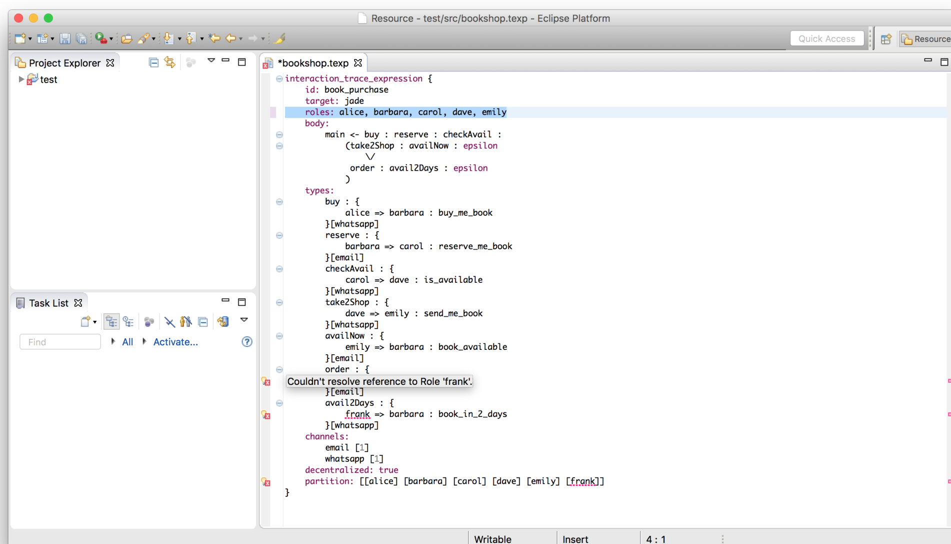

Figure 14.6. Error: After having removed the role “frank”, we have an existence error.

EVENT TYPES EXISTENCE: In Figure 14.7 we remove an event type from the `types` field. After that, RIVERtools finds the corresponding use inside the terms consisting the protocol body, and it communicates to the programmer about the use of an event type undefined.

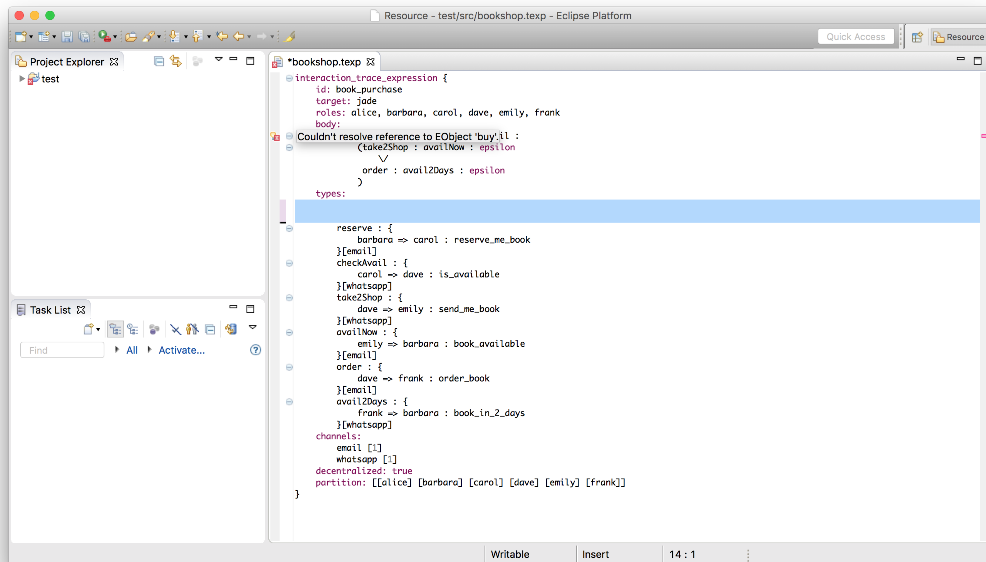

FIGURE 14.7. Error: After having removed the event type buy, we have an existence error.

14.4 *Tutorial: How to use RIVERtools*

Before ending this chapter with the final discussion, we report a brief tutorial on how to use the RIVERtools Eclipse plugin.

WHAT YOU ARE GOING TO INSTALL.

- SWI-Prolog
- Eclipse (with Xtext plugin)
- RIVERtools Eclipse plugin

14.4.1 How to install SWI-Prolog

Installing SWI-Prolog is very easy:

- On Linux:
  1. sudo apt-get install software-properties-common
  2. sudo apt-add-repository ppa:swi-prolog/stable
  3. sudo apt-get update

4. sudo apt-get install swi-prolog

- On Windows
  - Download either 32bit or 64bit from `http://www.swi-prolog.org/download/stable`
- On MacOSX (using Homebrew[12])
  - brew install swi-prolog

14.4.2 How to install RIVERtools Eclipse plugin

1. Link for downloading last Eclipse IDE: `https://www.eclipse.org/downloads/packages/installer`
2. Install Xtext plugin on Eclipse:
   a) Choose Help -> Install New Software... from the menu bar and Add...
   b) Insert `http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/`. This site aggregates all the necessary and optional components and dependencies of Xtext.
   c) Select the Xtext SDK from the category Xtext and complete the wizard by clicking the Next button until you can click Finish.
   d) After a quick download and a restart of Eclipse, Xtext is ready to use.
3. Link for downloading RIVERtools plugin: `https://github.com/AngeloFerrando/website/raw/master/assets/rivertools/plugin/rivertools.zip`
4. Install RIVERtools plugin:
   a) Choose Help -> Install New Software... from the menu bar and Add...
   b) Choose Archive...
   c) Select the rivertools.zip file downloaded previously.
   d) Select TExp Feature from the list ("group items by category" must be unchecked) and Next -> Finish.
   e) After a quick download and a restart of Eclipse, the plugin is ready to use.

14.4.3 How to use RIVERtools plugin (through an example)

1. Create a new project (a General project, not a Java Project) and call it MyPingPongExample.

[12] `http://mxcl.github.io/homebrew/`

2. Inside the created project, create a new file and call it pingpong.texp.

3. An Eclipse popup should ask you if you want to configure the project with Xtext: say yes. If Eclipse does not ask you, right-click on the project folder -> Configure -> Convert to Xtext project...

4. Now you can define trace expressions with the full support of the RIVERtools plugin.

Now you are ready to define inside RIVERtools a variation of the ping pong example presented previously in this chapter. Specifically, in this example the number of `ping` is equal to the number of `pong`.

```
interaction_trace_expression{
        id: pingpong
        target: jade
        roles:
                alice$SenderAgent('bob', 'ping', '5')$
                bob$ReceiverAgent('alice', 'pong', '5')$
        types:
                ping : { alice => bob : ping }
                pong : { bob => alice : pong }
        body:
                main <- pingpong * main
                pingpong <-
                        (ping : (pingpong \/ epsilon)
                                * pong : epsilon)
}
```

Since Eclipse is using the RIVERtools plugin to analyze the trace expression, all syntax and types errors are promptly indicated to the developer. As presented before in this chapter, we are interested in automatically generating the code that will be used for verifying our multiagent system. When no errors are found from RIVERtools, a new folder src-gen is created. Inside this folder we can find the Java and Prolog code that have been automatically generated through the compilation of the trace expression.

14.4.4 How to verify a MAS implemented in JADE

1. Link for downloading ping pong MAS example: `https://github.com/AngeloFerrando/website/raw/master/assets/rivertools/jadeconnector/example/pingpongmas.zip`

2. Import the example into Eclipse:
    a) File -> Import... -> Existing Projects into Workspace.
    b) Select archive file and select pingpongmas.zip downloaded previously, then finish.

3. Inside the JADE project example you have two agents `SenderAgent` and `ReceiverAgent`. Note that you have already indicated these classes inside pingpong.texp.

4. Copy Pingpong.java from the MyPingPongExample project to this project.

5. Run it. An exception will be thrown ("SWI_LIB environment variable not defined")

6. How to set the SWI_LIB environment variable
   a) Right-click on Pingpong.java (the one inside MyPingPongMAS) -> Run as -> Run Configurations...
   b) Pass to the Environment tab and then New...
   c) Set the name to SWI_LIB
   d) Set the value to the path to the SWI-Prolog library. For instance, on MacOSX is something like: /opt/local/lib/swipl-<version>/lib/x86_64-darwin15.0.0/

7. The last thing you need to do before running the MAS is to change the path to pingpong.pl (line 27 in Pingpong.java). You have to set it to the absolute path corresponding to the file generated in MyPingPongExample project.

8. You are now ready to execute the MAS with the automatically generated RV monitor.

9. Right-click on Pingpong.java (inside MyPingPongMAS) -> Run as -> Java Application

The `SenderAgent` and `ReceiverAgent` are parametric. In particular, the third parameter says how many messages the agent should send. In order to better observe if the monitor is working, you can introduce a wrong behaviour in the receiver agent. Instead of passing `5` as third argument to both the agents, you pass `5` to alice and `6` to bob. After that, you will observe the monitor printing an error message when the sixth `pong` will be observed.

RIVERtools is an open source project and is available on github. Link for cloning the project: `https://github.com/AngeloFerrando/trace_expression_plugin_eclipse.git`

14.5 Discussion

In this chapter we have presented RIVERtools, an IDE that can be used for the integration of the trace expression formalism with any possible target system. Thanks to the presence of connectors that handle all the domain dependent

issues, we have showed that RIVERtools allows focusing on a more abstract level leaving all technical details to the connector implementation.

We have presented the RIVERtools general structure and we have analyzed its possible use through an example involving a `book-shop` scenario developed as a MAS in JADE. Since the development of RIVERtools is very recent, we have not yet stress-tested it on a real, complex case study. The results obtained on the toy examples used for preliminary testing are promising and we plan to identify some challenging scenario where we need to fully exploit the expressive power of trace expressions. This will allow us to evaluate the benefits of using RIVERtools, as well as its limitations, in a systematic way.

15 Case Study

"My doctor told me to stop having intimate dinners for four.
Unless there are three other people."
- Orson Welles

Remote Patient Monitoring (RPM) enables physicians to perform diagnosis and/or treatment remotely through sensors connected via a communication network. Dependability and flexibility are recognized as two key technological requirements for RPM take off. In this chapter we present a case study in the medical field where trace expressions are exploited for modeling protocols as complex and sophisticated as those used in healthcare. In this scenario, trace expressions are used both for static verification, following the approach introduced in Chapter 12, and for driving the agent behavior, according to the approach presented in (Ancona et al., 2015a). Support to protocol-driven code generation may attain flexibility, fault tolerance and removal, and fault prevention in safety-critical systems, as discussed in Section 5.1. In the same way as trace expressions have been employed for protocol-driven behavior, they could have been used – with minor modifications – for RV and DRV as well. The choice of exploiting – in this scenario – trace expressions for a goal different from RV and DRV allowed us to demonstrate their full potential, which goes far beyond the aspects dealt with in this Thesis.

The contents of this chapter are published in (Ancona, Ferrando, and Mascardi, 2018b; Ferrando, Ancona, and Mascardi, 2016)

15.1 Introduction

Remote Patient Monitoring (RPM (Bayliss et al., 2003)) allows patients to use mobile medical devices to perform routine tests at home and get an immediate feedback from the device itself, and/or automatically send the test data to healthcare professionals to get a feedback in real time.

Even if some studies raised doubts on the RPM effectiveness (Chaudhry et al., 2010), others, such as the Whole System Demonstrator Programme (WSD (UK Department of Health, 2011)), had more promising outcomes. WSD involved 6191 patients who were monitored for 12 months between May 2008 and December 2009, across three sites in Newham, Kent and Cornwall. WSD showed that RPM is associated with lower mortality and emergency admission rates (Steventon et al., 2012). Positive results are also presented in (Kraai et al., 2016) and (Yoo et al., 2014), among the others. The first study showed that using RPM in patients with heart failure is safe and can reduce heart failure-related visits to the outpatient clinic, keeping care accessible, even if the cost is higher than that of conventional care. The second shows that a centralized telecare management, coupled with automated symptom monitoring, appears to be a cost effective intervention for managing pain and depression in cancer patients. Other success stories are presented in (Gensini et al., 2017), along with an analysis of the barriers to the RPM diffusion.

The take off of RPM heavily depends on financial, organizational, legal, and psychological aspects (Wallace et al., 2017; Wood, Boulanger, and Padwal, 2017): initial and maintenance costs should justify the adoption of RPM and the changes it causes to the healthcare business model, legal issues should be properly faced, patients and physicians should be willing to accept new technologies in their daily work and lives.

From a technological point of view, dependability is a key factor for RPM successful adoption. As observed in (Jezewski et al., 2016) w.r.t. medical cyber-physical systems for telemonitoring pregnancy at home,

> *dependability has many dimensions like: reliability, security, safety, privacy, and trust that must be resolved and assured through careful design verification, validation and final certification processes. [...] Dependable cyber-physical systems can be achieved if appropriate verification, validation and certification processes are conducted [...].*

This position is consistent with that discussed in (Winikoff, 2017), according to which trust is one of the main requirements in Human-Computer and Human-MAS interaction and it can be made stronger by a-priori verification.

Besides dependability, another enabling factor for RPM is flexibility, meant as “the ability to be easily modified”. In order for a software system to be easy and cheap to maintain, this ability should require limited or no intervention from human designers and developers. In other words, the system should be able to autonomously and dynamically adapt its functioning to changing environment conditions. While this is important for many application domains,

it becomes of paramount importance in a RPM setting as discussed in (Kucher and Weyns, 2013).

One way to achieve flexibility is adopting a multiagent approach where agents are capable of self-adaptation, whereas dependability can be attained in different ways, including preventing and removing faults during the system development and use by exploiting software engineering tools and methods, testing, and formal verification techniques (Avizienis et al., 2004, Sec. 5). If we merge the two approaches, RPM systems can be made flexible and dependable by designing and implementing them as self-adaptive MAS where agents development is supported by IDEs, and agents are driven by specifications that can undergo testing, static analysis including model checking, and runtime monitoring.

In this section we first provide a gentle introduction to the practical use of trace expressions for RPM via examples taken from (Bottrighi et al., 2010). After that, we describe a complex protocol for managing hypoglycemia in the newborns which extends the work presented in (Ferrando, Ancona, and Mascardi, 2016) by exploiting parameters in the model.

15.1.1 A Jason Framework Supporting Agents Driven by Parametric Trace Expressions

According to (Ancona et al., 2015a), in order to support a protocol-driven approach to agent programming a *generate* function for identifying the allowed actions for moving from the current state of the protocol to the next one must be provided. *Generate* is built on top of the *next* function implementing the δ transition relation. Each agent is characterized by a *select* policy to select the action to perform among the allowed ones, and a *react* policy to react to perceived events. Two more policies state how to manage *unexpected* events and which *cleanup* actions to perform before switching from the currently executing protocol to the new one. Self-adaptation is performed by means of a protocol switch triggered by the reception of "switch requests" sent by "super-agents" or "controllers". Agents can request protocol switches to themselves. Agents are also able to *project* a global description of a protocol involving many agents onto a local version by keeping only the events where they are involved in.

The protocol-driven interpreter implements a cycle where it first checks if there is a protocol switch request and if it can be managed in the current state of the protocol. If yes, and if the protocol switch sender is a system controller, a protocol switch is performed after some cleanup operations. If no protocol switch is foreseen in that moment, the protocol-compliant actions are generated and one of them is selected for being executed. The environment representation and knowledge base are updated accordingly and the protocol moves to the next state. In case the perceived event was not foreseen by the protocol, it is managed according to the unexpected policy.

In the framework presented in (Ancona et al., 2015a) and extended in (Ferrando, Ancona, and Mascardi, 2016) to cope with generic events, the protocol-

driven engine was implemented in “Jason Prolog”, namely the Prolog version natively supported by Jason. W.r.t. SWI-Prolog, the “Jason Prolog” shows some syntactic differences which, even if minor, have to be carefully considered when modeling the protocol.

The introduction of parameters into trace expression required to heavily exploit SWI-Prolog technical features for moving from the representation of a variable as a logical variable in the protocol, to an internal explicit representation whose associated values could be manipulated by the interpreter. In other words, the variable unification mechanism which is normally hidden to the Prolog programmer had to be made explicit in order to implement the `match` function.

Porting this complex management of logical variables into “Jason Prolog” was not feasible, and the Jason framework for self-adaptive agents driven by parametric protocols was re-implemented by allowing Jason to communicate with SWI-Prolog in order to use operations on trace expressions like `match`, `next`, `generate`.

Making Jason and SWI-Prolog communicate raised some technical details. If we only had to verify that the Jason MAS behaves according to some given protocol, and both the protocol and the verification engine were implemented in SWI-Prolog, the problem would be simpler. In fact, each time an event is observed, Jason should pass the observed event to the SWI-Prolog “verification module” and ask to check if the event is consistent with the current state of the protocol. The SWI-Prolog module should return the answer computed by `next`, which is either true or false.

In the protocol-driven setting, the answer generated by SWI-Prolog is the result computed by `generate`: it represents the set of all possible events that the protocol-driven agent can choose. While translating true/false from SWI-Prolog to Jason is easy, translating a set of events is more challenging from a technical point of view.

Making Jason and SWI-Prolog communicate required to develop some adhoc Prolog code both on the Jason side (300 lines of Java code), and on the SWI-Prolog side (90 LOC).

15.1.2 Modeling Clinical Guidelines

Clinical guidelines (GLs) specify the best way to take care for people with specific pathological conditions. The identification and standardization of GLs is fundamental for making RPM possible, as discussed for example in (Bonnell and Mittal, 2013) w.r.t. patients with cardiac implantable electronic devices. In (Bottrighi et al., 2010) more than twenty GLs have been modeled using LTL and then verified by integrating a computerized GL management system with a model-checker.

In the sequel we provide examples of GLs analyzed in (Bottrighi et al., 2010) modeled using trace expressions. $\forall_{run}.(run \in [\![\tau]\!])$ identifies a property that must hold for all the traces in $[\![\tau]\!]$ and corresponds to an LTL property characterized by an outer “Globally” operator, whereas $\exists_{run}.(run \in [\![\tau]\!])$ identifies a

property that must hold for at least one event trace in $[\![\tau]\!]$ and corresponds to an LTL property characterized by an outer "Eventually" operator. The negation $\neg$ is not part of the trace expressions syntax; it is a shortcut for identifying all the events that do not match a given event type. More formally, given $\mathcal{E}$ the fixed universe of modelled events and ϑ an event type (hence, a set of events), $\neg\vartheta = \mathcal{E} \setminus \vartheta$.

The description, comment and relevance of each example are quoted from (Bottrighi et al., 2010). The specification of the other GLs presented in that paper can be found in Appendix A.

Example of structural property.
Verify that neurological deficit is always present (ischemic stroke GL).

$$\forall_{run}.(run \in [\![\tau]\!])$$

$$\tau = (neurological_deficit{:}\tau) \vee \epsilon$$

Comment: The property is true if neurological deficit is always present, in all the states of all possible runs.
Relevance: The ischemic stroke GL is not applicable to a patient with no neurological deficit.
Trace expression representation: if the event type *neurological_deficit* appearing in τ is the set containing only one event *nd*, then the semantics of τ is the set of traces $\{\epsilon, nd, nd\ nd, nd\ nd\ nd, \ldots, nd^{\omega}\}$. The property that for each trace the neurological deficit is always present, is correctly modelled by τ.

Example of medical validity property.
Verify that whenever hepatic encephalopathy is present, diuretics are not administered.

$$\forall_{run}.(run \in [\![\tau]\!])$$

$$\tau = (liver_encephalopathy{:}\tau_1) \vee (\neg liver_encephalopathy{:}\tau) \vee \epsilon$$

$$\tau_1 = (\neg diuretics_administration{:}\tau_1) \vee \epsilon$$

Comment: Diuretics are contraindicated in hepatic encephalopathy.
Relevance: Diuretics can worsen the liver perfusion and precipitate the encephalopathy or worsen its severity.
Trace expression representation: if the event type $liver_encephalopathy$ contains le only, the event type $diuretics_administration$ contains da only, and there are no other events modelled in the system, then

$$[\![\tau]\!] = \{\epsilon, da, da\ da, \ldots, da^{\omega}, le, da\ le, da\ da\ le, \ldots, da\ \ldots\ da\ le, le\ le, da\ le\ le, \ldots\}$$

. The property that that for each trace, if le belongs to the trace, than it is never followed by da, is correctly modelled by τ.

Example of contextualization property.
Verify that there is at least one run in which the Computed Tomography (CT) scanner is not used (ischemic stroke GL).

$$\exists_{run}.(run \in [\![\tau]\!])$$

$$\tau = (\neg CTscan{:}\tau)\vee(CTscan{:}\epsilon)\vee\epsilon$$

Comment: If this condition holds, the GL (or, at least a part of it) can be applied also in hospitals where the CT scanner is not available.
Relevance: The CT scanner is very important in some cases but not always accessible.
Trace expression representation: let us suppose that event type $CTscan$ contains only the event ct, and the other events are identified by oth in the event trace.
$[\![\tau]\!] = \{\epsilon, oth, oth\,oth, \ldots, oth^{\omega}, ct, oth\,ct, oth\,oth\,ct, oth\,oth\,oth\,ct, \ldots, oth \ldots oth\,ct\}$.
The property that there exists a trace where no CT scan is needed, is correctly modelled by τ. To be precise, there are infinite traces satisfying the property.

15.1.3 Modeling Management of Hypoglycemia in the Newborns

In this section we briefly recall the protocols for managing hypoglycemia in the newborns introduced in (Ferrando, Ancona, and Mascardi, 2016) and we present their parametric version.

The human beings involved in this case study are doctors and newborns suffering from hypoglycemia. Each of them is associated with a protocol-driven agent. Each baby should be equipped with sensors able to change the protocol-driven agent knowledge base after perception of sensory input such as *temperature*, *heartbeat*, *pressure*, *O_2 saturation*, *movement*, and so on.

Hypoglycemia management does not require a constant presence of a doctor but needs an ongoing monitoring of some vital parameters. The protocol-driven agent associated with newborns might need to communicate something to the patient's parents by printing some message onto a screen positioned near to the newborn. Parents can follow the monitoring process and the intervention instructions like, for example, the request to inject a dose of glucose solution.

The doctor's agent is driven by a single protocol characterized by three mutually exclusive situations:

- Ok situation (*Ok* sub-protocol): if the doctor agent receives a message reporting that the glucose level in the blood of its patient is ok, then things are going on in the right way: the agent goes on monitoring;
- Severe situation (*SevereProblem* sub-protocol): if the doctor agent receives a message reporting that the glucose level in the blood of its patient is too low, then it sends a protocol switch request to the patient and continues to monitor;
- Possibly critical situation (*SwitchedToSeverity1* sub-protocol): if the doctor agent receives a message reporting that the patient has switched its protocol to severity1 two thing may happen:
  - if the agent either receives a message reporting that the patient has tremors or he/she is irritable, then it is up to the doctor's agent to decide whether asking to the patient to continue the monitoring activity, or moving to a critical situation state (switch to *severity2* protocol);

- if the doctor's agent receives a message from the patient (or from his parents) reporting an intervention request, then the agent communicates this request to the "real" doctor using the screen.

In both cases, the agent can move to the "standard" monitoring state modeled by the trace expression associated with the *DoctorPatientProtocol*.

The patients' agents are driven, in each time instant, by one of the three protocols below, each representing the "dual" of the corresponding doctor's protocol:

- *standard protocol*, which models the situation where the newborn has no symptoms of the disease;
- *severity 1 protocol*, associated with the lowest severity level of the disease;
- *severity 2 protocol*, associated with the highest severity level of the disease.

Parametric extensions

Doctor protocol. We added two parameters to the Doctor protocol: *Hours* and *Threshold*. They are used to dynamically represent the baby's hours of life and the minimum glucose level respectively, and allow us to introduce parametricity via a simple example.

The *Hours* variable is used to track of the baby's hours of life. With respect to the non-parametric protocol version, in the parametric version we can also model time. In this way, during the monitoring, the doctor can decide which is the correct glucose threshold, modeled by the *Threshold* variable, on the basis of the current baby's hours of life.

The parametric protocol has one more branch in the main trace expression *T1*. This branch is represented by the trace expression *Age*, used to keep track of the information related to the baby's hours of life. Thanks to the new construct *var*, we can instantiate the *Hours* variable which will be set at runtime to the proper current value passed as content of the *msg_age* message. After that, *Hours* is used inside the *msg_threshold_send* event type in order to decide the correct threshold which has to be communicated to the patient. The other parts of the protocol do not change.

Note that the *Threshold* variable does not appear in the Doctor protocol because its value is used only by the Patient's Protocol. In particular, the threshold value is set and passed to the Patient's Protocol in the *msg_threshold_send* event type.

We remind that the agent associated with the doctor has the power to request a protocol switch to the patient's agent modeled by the *switch_request(Sender, Receiver, NewProtocolIdentifier)* event type.

```
trace_expression(hypoglycemia_doctor_protocol, T) :-
  Ok = msg_ok_glucose(var(1), var(2)):OkOrProblem,
  Problem = msg_too_low_glucose(var(1), var(2)):SwitchSeverity2,
```

```
  OkOrProblem = (Ok \/ Problem),
  SwitchSeverity2 = (switch_severity2(var(2), var(1)):T1) \/
                   (msg_continue_monitor(var(2), var(1)):T1),
  Tremors = msg_tremors(var(1), var(2)):SwitchSeverity2,
  Irritability = msg_irritability(var(1), var(2)):SwitchSeverity2,
  InterventionRequest = intervention_request(var(1), var(2)):
                   print_intervention_request(var(1), var(3)):epsilon,
  MsgSeverity1 = msg_severity1(var(1), var(2)):epsilon,
  Age = var(Hours, msg_age(var(1), var(2), Hours):
                   msg_threshold_send(var(2), var(1), Hours):Age),
  T1 = (Age | ((OkOrProblem |
                   (MsgSeverity1 * (Tremors \/ Irritability)))
                   | InterventionRequest)),
  T = finite_composition(|, T1, [m(var(1),[]),m(var(2),[]),m(var(3),[])]).
```

Patient standard protocol. Also this protocol has been extended by introducing the construct *var* to manage the hours of life and the glucose level threshold. In particular, by introducing the trace expression *Age* where we set the *Hours* variable, it is possible to communicate the age to the Doctor and to set the *Threshold* variable respectively:

- The event which matches the *my_age* event type is a perception containing the information about the baby's hours of life. In this way, when this event type is matched, the variable *Hours* is set to the correct value.
- After that, a message to the Doctor containing the *Hours* variable, fixed to the proper value, is sent.
- Finally, a message from the Doctor is received (*msg_threshold_recv* event type) setting the *Threshold* variable to the correct value decided by the doctor.

Once the *Age* part is consumed, the protocol continues as in the non-parametric case. The only difference is that we can use the *Threshold* variable to influence the *ok_glucose* and *too_low_glucose* event types. Indeed, since the glucose level used as threshold in the protocol dynamic, we can use it when we observe the events taken by sensors, to decide if the glucose level is under control or not.

```
trace_expression(hypoglycemia_protocol_standard, T) :-
  Ok = ok_glucose(var(1), Threshold):
             msg_ok_glucose(var(1), var(2)):Age,
  Problem = too_low_glucose(var(1), Threshold):
             msg_too_low_glucose(var(1), var(2)):
             ((msg_continue_monitor(var(2), var(1)):Age) \/
              (switch_severity2(var(2), var(1)):epsilon)),
  OkOrProblem = (Ok \/ Problem),
  T1 = (Age |
             (tremors(var(1)):epsilon) |
             (irritability(var(1)):epsilon) |
```

```
              (convulsions(var(1)):epsilon) |
              (apnea(var(1)):epsilon) |
              (irregular_breathing(var(1)):epsilon)),
  Age = var(Hours, my_age(var(1), Hours):
              msg_age(var(1), var(2), Hours):
              var(Threshold, msg_threshold_recv(var(2), var(1), Threshold):
              OkOrProblem)),
  T = finite_composition(|, T1, [m(var(1), []), m(var(2), [])]).
```

Severity 1 and Severity 2 protocols. Also in these protocols, two variables *Hours* and *Threshold* have been added and used in the same way as for the patient standard protocol.

```
trace_expression(hypoglycemia_protocol_severity1, T) :-
  Ok = ok_glucose(var(1), Threshold):
                   msg_ok_glucose(var(1), var(2)):Age,
  Problem = too_low_glucose(var(1), Threshold):
                   msg_too_low_glucose(var(1), var(2)):
                   ((switch_severity2(var(2), var(1)):epsilon) \/
                    (msg_continue_monitor(var(2), var(1)):Age)),
  Msg = msg_severity1(var(1), var(2)):epsilon,
  Tremors = severe_tremors(var(1)):msg_tremors(var(1), var(2)):
                   ((msg_continue_monitor(var(2), var(1)):Tremors) \/
                    (switch_severity2(var(2), var(1)):epsilon)),
  Irritability = severe_irritability(var(1)):msg_irritability(var(1), var(2)):
                   ((msg_continue_monitor(var(2), var(1)):Age) \/
                    (switch_severity2(var(2), var(1)):epsilon)),
  OkOrProblem = (Ok \/ Problem),
  T1 = Msg * (Age | Tremors | Irritability |
                   (convulsions(var(1)):epsilon) |
                   (apnea(var(1)):epsilon) |
                   (irregular_breathing(var(1)):epsilon)),
  Age = var(Hours, my_age(var(1), Hours):
                   msg_age(var(1), var(2), Hours):
                   var(Threshold,
                     msg_threshold_recv(var(2), var(1), Threshold):OkOrProblem)),
  T = finite_composition(|, T1, [m(var(1), []), m(var(2), [])]).

trace_expression(hypoglycemia_protocol_severity2, T) :-
  Ok = ok_glucose(var(1), Threshold):
             msg_ok_glucose(var(1), var(2)):Age,
  Problem = too_low_glucose(var(1), Threshold):
             intervention_request(var(1), var(2)):epsilon,
  T1 = intravenous_inj_glucose_sol(var(1), var(3), 10):
             (Ok \/ Problem),
  Age = var(Hours, my_age(var(1), Hours):
             msg_age(var(1), var(2), Hours):
             var(Threshold, msg_threshold_recv(var(2), var(1), Threshold):T1)),
  T = finite_composition(|, Age, [m(var(1), []), m(var(2), []), m(var(3), [])]).
```

15.2 Experiments

We carried out two experiments aimed at showing trace expressions support to a priori verification and self-adaptation. A third experiment measured the implementation performances.

15.2.1 A priori verification (attains fault tolerance and removal)

We exploited the model checking mechanism presented in Chapter 12 for verifying that the clinical guidelines described in Section 15.1.2 and in Appendix A respect the corresponding LTL properties described in (Bottrighi et al., 2010).

To this aim, the trace expressions were translated into Büchi automata. There was no need to rewrite them, as the trace expressions presented in Section 15.1.2 have the same expressive power than LTL and Büchi automata and no over-approximation is required. Then, SPIN was used to perform the model checking stage.

Following this approach we were able to check, for example, that the trace expression modeling the clinical guideline "verify that neurological deficit is always present" verifies the LTL property $\langle \forall run, \Box(neurological\ \ deficit = present) \rangle$.

This check was done for all the guidelines presented in this chapter. The result is that **all the trace expressions modeling medical guidelines presented in this chapter satisfy the corresponding LTL properties presented in (Bottrighi et al., 2010).**

This result shows the suitability of trace expressions to model general behavioral patterns and verify that the model respects some mandatory, safety-critical conditions before using it to drive the behavior of agents in a real system.

15.2.2 Self-adaptation (attains flexibility and fault tolerance and removal)

The parametric trace expressions presented in Section 15.1.3 were used to drive the behavior of agents in a Jason MAS. Sensory input was simulated. In this suite of functional tests, we run 20 experiments with a maximum of 10 patients, 1 doctor, 11 screens that agents use for interacting with their human users (one for each patient and one for the doctor), 10 sensors.

For each test we manually checked the conversation among agents, in particular w.r.t. the occurrence of protocol switch (the means provided by trace expressions to support self-adaptation), and all the conversations were consistent with the hypoglycemia in the newborns protocol. Since a MAS involving protocol-driven agents is safe by construction (Ancona et al., 2015a), as agents are forced to behave according to the given protocol, **consistency is the only result we could obtain**: functional tests just confirmed this fact.

Below, we report one selected conversation which is easy to follow, to exemplify the interface of the Jason framework presented in Section 15.1.1 and the manual check process.

Figure 15.1 shows a fragment of an interaction among agents *patient1*, *patient2* and *doctor1*: after *patient1* perceives the event *my_age(1)* (meaning that the baby *patient1* is in charge for, is 1 hour old), it sends the corresponding message *msg_age(1)* to *doctor1*. The same happens with *patient2*, whose baby is 2 hours old. Agent *doctor1* receives both messages and moves to a new state, where this information is taken into account.

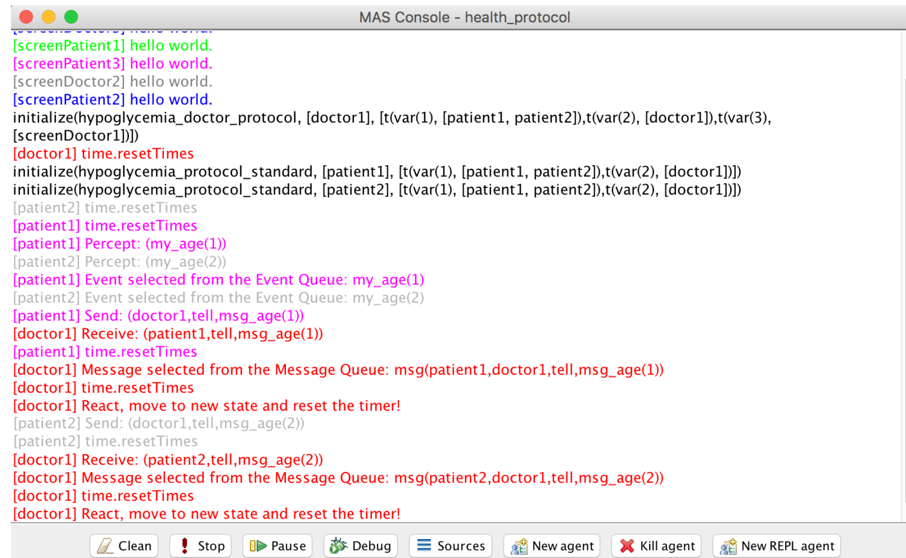

FIGURE 15.1. Agents *patient1* and *patient2* sending the hours of life to agent *doctor1*.

Given the hours of life, *doctor1* chooses as glucose level threshold the value *28* for the baby monitored by *patient1* and sends this value to it (Figure 15.2).

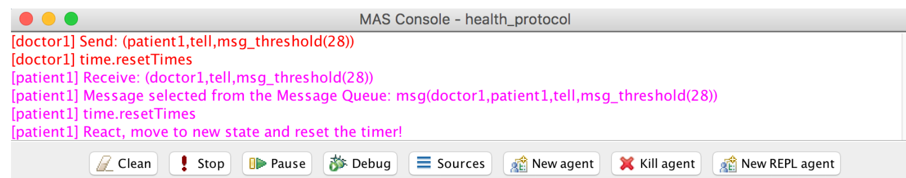

FIGURE 15.2. Agent *doctor1* sending the threshold to *patient1*.

Figure 15.3 shows *patient1* perceiving the event *plasma_level_glucose(27)*. Since the glucose level is less than the threshold set by *doctor1*, the event *plasma_level_glucose(27)* matches the event type *too_low_glucose*. Consequently, a message telling that the perceived glucose level is too low is sent to *doctor1*.

When agent *doctor1* receives the message about the low glucose level, it sends a protocol switch request to *patient1*. When *patient1* receives this message, it switches to the severity 2 protocol and starts behaving according to it (Figure 15.4).

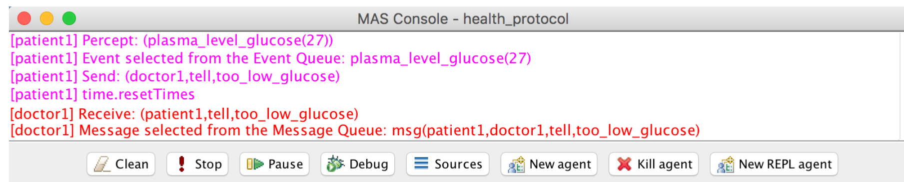

FIGURE 15.3. Agent *patient1* perceiving a low level of glucose.

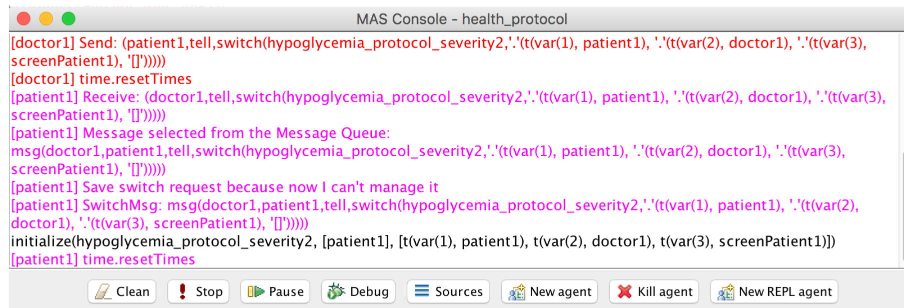

FIGURE 15.4. Agent *doctor1* asking for a protocol switch to *patient1*.

Figure 15.5 shows that, as specified by the severity 2 protocol, agent *patient1* sends a message to its screen in order to display the request to inject a 10% glucose solution, informing in this way the baby's parents.

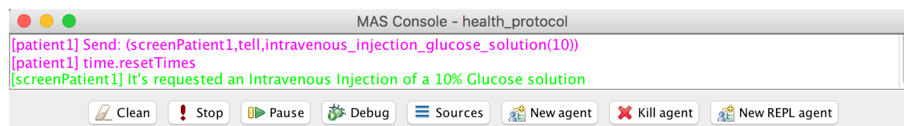

FIGURE 15.5. Agent *patient1* displaying the need of an intravenous injection.

Finally, since the baby glucose level is still decreasing, agent *patient1* sends a request to agent *doctor1* to inform its user, namely the real doctor, to intervene. When agent *doctor1* receives this message, it displays it on its screen to involve the doctor in the decision process (Figure 15.6).

15.2.3 Performances

The last suite of experiments was devoted to assess the performances of the Jason framework presented in Section 15.1.1. We run 20 tests for each of the following configurations, identified by letters from **a** to **i** in Table 15.1: one doctor with ten patients, one doctor with fifty patients, one doctor with one hundred patients, two doctors with ten (resp. fifty, one hundred) patients each, and five doctors with ten (resp. fifty, one hundred) patients each. Age and dynamics of the situation evolution of patients were randomly generated for each run.

The number of tests we run was 180, with a number of agents – including sensors – ranging from 22 and 1010. For each configuration we computed the average number of messages sent by each doctor and the average number of

```
MAS Console - health_protocol
[patient1] Percept: (plasma_level_glucose(25))
[patient1] Event selected from the Event Queue: plasma_level_glucose(25)
[patient1] Send: (doctor1,tell,intervention_request)
[patient1] time.resetTimes
[doctor1] Receive: (patient1,tell,intervention_request)
[doctor1] Message selected from the Message Queue: msg(patient1,doctor1,tell,intervention_request)
[doctor1] time.resetTimes
[doctor1] React, move to new state and reset the timer!
[doctor1] Send: (screenDoctor1,tell,intervention_request(patient1))
[screenDoctor1] The patient patient1 needs of your intervention
Clean  Stop  Pause  Debug  Sources  New agent  Kill agent  New REPL agent
```

FIGURE 15.6. Agent *patient1* asking *doctor1* to intervene.

messages sent by each patient in 5, 20 and 100 seconds to identify degradations in the communication performances.

We could not measure the system performances via experiments like "how many seconds are needed by the system to reach some given configuration or to terminate" because the protocol could go on forever (for example, if the patient is stable) and there is no final configuration to reach, as different runs can lead to different final states.

Conf.	**Doct.** (*D*)	**Pat. per doct.** (*P*)	**Tot. ags.**	**Sent in 5s (doct.)**	**Sent in 5s (pat.)**	**Sent in 20s (doct.)**	**Sent in 20s (pat.)**	**Sent in 100s (doct.)**	**Sent in 100s (pat.)**
a	1	10	22	4.5	1	16	3.5	88	14.8
b	1	50	102	5.5	0.7	18	1.9	100	7.2
c	1	100	202	4	0.4	18	1.7	80.5	7
d	2	10	44	4	1.4	17	3.2	81.5	17
e	2	50	104	4.8	0.6	22	2.1	102.2	9.5
f	2	100	404	2	0.35	9	2	42.5	6.8
g	5	10	110	4.7	1	17	3.9	89.3	18.7
h	5	50	510	0.6	0.36	2	1.3	12	5.8
i	5	100	1010	0.6	0.15	2.2	0.75	10	3.2

TABLE 15.1. Experiments

Table 15.1 shows that, for a given number of doctors D and a given amount of time T, the number of messages sent by each patient in T decreases with the increase of the patients' number. Doctors tend to maintain a more constant rate of sent messages as long as the total number of agents is 200 or less (configurations **a**, **b**, **c**, **d**, **e**, **g**). When the total amount of agents is 400 or more (configurations **f**, **h**, **i**), the average number of messages that doctors send in a given amount of time T clearly decreases. This result is the expected one and can be easily explained: protocol-driven agents run on the same machine and the more the agents, the less frequent each agent is scheduled and can send messages.

Although having one doctor agent allowed to send only 10 messages every 20 seconds (configuration **i**) may raise serious problems when quick responses are needed, we should consider that the stress-test we performed is not realistic for two reasons:

1. in a real setting, we expect that no doctor is in charge of a high number of newborn babies that require constant remote monitoring: configurations **c**, **f**, and **i** are hopefully unrealistic;

2. if a real protocol-driven MAS were built following the approach discussed in this chapter, each doctor's agent would run on the actual doctor's machine, and the same would happen for each patient's agent: with one doctor and fifty patients, performances of configuration **b** represent a lowerbound for the actual performances.

Given the above considerations, we believe that **a system where a doctor agent can send about one message every second (one message every P seconds to each of its P patients) is compliant with a RPM setting where P is low and intervention in case of need is made by human beings**, like the hypoglicemia in the newborns one.

15.3 Discussion

In this chapter we presented a challenging case study in the RPM field. We showed how trace expressions are a suitable formalism for representing medical guidelines, and how their self-adaptivity can be exploited in this scenarios.

The ability of trace expressions to self-adapt (hence attaining fault tolerance and removal) has been demonstrated in Section 15.1.3 by the case study design and implementation (protocol switch is triggered by the effect of monitoring the environment, according to the perspective on self-adaptivenness proposed in Weyns, 2018) and in Section 15.2.2 where we showed that adaptations take place as expected. For example, we have correctly observed that *"when agent doctor1 receives the message about the low glucose level, it sends a protocol switch request to patient1. When patient1 receives this message, it switches to the severity 2 protocol and starts behaving according to it."*

The ability of trace expressions to undergo static verification (hence attaining fault prevention) is demonstrated in Section 15.1.2 and in Appendix A by the medical guidelines design, and in Section 15.2.1 by their verification.

Part VII

Discussion

In this part we compare the trace expression formalism with the formalisms presented in the state of the art at the beginning of the thesis. Finally, we conclude the work with the final comments, conclusions and future work.

16 Comparison

16.1 Trace Expressions VS State of the art

Trace expressions have been continuously refined and consolidated during the last 5 years (Master's degree and Ph.D. program) (Ancona, Barbieri, and Mascardi, 2013; Ancona, Ferrando, and Mascardi, 2016; Ancona et al., 2014, 2015b, 2016; Briola, Mascardi, and Ancona, 2014a,b; Mascardi, Briola, and Ancona, 2013). Their most recent extensions involve the introduction of parameters (Chapter 6) and probabilities (Chapter 7), the safe decentralization of the MAS monitoring process (Chapter 9 and 10), and the implementation of the RIVER-tools IDE (Chapter 14).

In the following analysis, we take the same ten features into account as in the state of the art chapter (Chapter 5):
– **Modelled issues**: interaction protocols from a global perspective but also more general behavioural patterns involving many parties.
– **Modeling approach**: parametric trace expressions are built on top of the "event" and "event type" notions; an event is something which takes place in the environment and which can be either generated or observed by the agents. It may be a communicative event like in (Ancona et al., 2015a) or any other event like the perception of some value from a sensor, the observation of some phenomenon in the environment, the expiration of a deadline (Ancona et al., 2015b). Trace expressions can be combined using *concatenation*, *intersection*, *union*, and *shuffle* operators reaching the expressive power of non context-free grammars, higher than that of FSM which can recognize regular grammars.
– **Integrated development environment**: a prototype tool to design trace expressions and perform static checks and operations on them is presented in Chapter 14.
– **Parametricity**: in Chapter 6 we showed how trace expressions are extended with parameters by introducing variables that are substituted with data values at runtime, when events are matched during monitoring. This extension advances previous proposals (Ferrando, 2015; Mascardi and Ancona, 2013) and has also been adopted outside a MAS setting (Ancona et al., 2017a).
– **Probability**: in Chapter 7 we showed how to extend event types with probabilities in order to handle the absence of information during the runtime verification process.
– **Testing/simulation**: given a trace expression, all the traces of a given length compliant with that expression can be automatically generated, for further manual or automatic testing. Simulation is not supported.
– **A priori verification**: in Chapter 12, the translation of a trace expression into a Büchi Automaton in order to realize an automata-based model checking

is presented. Since trace expressions are more expressive than Linear Temporal Logic (LTL (Pnueli, 1977)), the translation generates an over-approximation of the trace expression leading to a sound procedure to verify LTL properties.
– **Runtime verification**: trace expressions and the previous notations they are based upon, have been conceived and implemented with runtime verification in mind. The verification mechanism is supported by both Jason (Bordini, Hübner, and Wooldridge, 2007) and JADE (Bellifemine, Caire, and Greenwood, 2007).
– **Self-adaptation**: in (Ancona et al., 2015a) a framework for top-down centralized self-adaptive MAS is presented; adaptive agents are protocol-driven and adaptation consists in runtime protocol switch.
– **Protocol enactability and protocol-driven code generation**: a working Jason interpreter for protocol-driven agents is presented in (Ancona et al., 2015a). No skeletal Jason code is generated: rather, thanks to the implemented interpreter plus some strategies that must be provided by the developer, the behaviour of the agents is entirely driven by the protocol itself. The problem of partial distribution of the protocol verification mechanism even when "standard" enactability conditions do not hold is faced in Chapter 9.
– **Case studies and applications**: the formalization of the FYPA protocol (Briola and Mascardi, 2011) using attribute global types, a predecessor notation of trace expressions, is presented in (Mascardi, Briola, and Ancona, 2013). FYPA (Find Your Path, Agent!) is a MAS implemented in JADE and used by Ansaldo STS for allocating platforms and tracks to trains inside Italian stations.

16.1.1 Comparison

Table 16.1 summarizes the analysis carried out in Chapter 5 and this chapter. As far as the **modelled issues** are concerned, GAIP stands for "agent interaction protocols from a global perspective" and GAIP+ denotes the possibility to model other aspects besides GAIPs. For the other issues apart from **applications**, ✔ means that an implemented support to the feature is provided whereas (✔) means that the feature is not supported but could be with limited effort (for example, algorithms have already been designed or code for similar purposes exist and should just be adapted). Finally, as far as **applications** are concerned, ✔ means that applications have been developed for external stakeholders such as enterprises, public administrations, research institutes, and that someone different from the authors is using them. By (✔) we mean that case studies and/or applications have been developed, maybe inspired by real problems, but none apart from the authors is using the approach.

Table 16.1 shows that the four approaches analyzed in Chapter 5 and trace expressions support most of the features we have considered.

Commitment machines, self-adaptive approaches, and parametric trace expressions are almost equivalent w.r.t. the analyzed features, each one surpassing the others under some respect and each suitable for the design, development and verification of a software system.

As far as trace expressions are concerned, they are currently also used

	BSPL	CM	HAPN	Self-adaptive	Trace expressions
First paper published in	2011	2001	2017	2004	2012
Modelled issues	GAIP	GAIP	GAIP	GAIP+	GAIP+
IDE		✔	✔	✔	✔
Parametricity	✔	✔	✔	✔	✔
Probability	(✔)	✔		✔	✔
Testing/simulation	(✔)	✔	✔	✔	(✔)
A priori verification	✔	✔	(✔)	✔	✔
Runtime verication	✔	✔	(✔)	✔	✔
Self-adaptation		✔		✔	✔
Enactment/code generation	✔	(✔)			✔
Applications	(✔)	✔	(✔)	✔	(✔)

Table 16.1. Comparison

for RV of object-oriented languages, s.t. Java (Ancona et al., 2017a) and of IoT architectures, specifically of Node-RED[1] (Leotta et al., 2018) and Node.js (Ancona et al., 2017b) where trace expressions have been successfully exploited for specifying the correct usage of API functions and a prototype is presented.

[1] https://nodered.org

17 Conclusions and Future Work

"Don't cry because it's over,
smile because it happened."
- Dr. Seuss

In this thesis, we presented all the work which has been done on runtime and static verification of multiagent systems during the 3 years of the Ph.D. program. We presented the trace expression formalism (Chapter 4) with its parametric and probabilistic extensions (Chapter 6 and Chapter 7) and we showed different scenarios of possible use for each one of them. In this Chapter we will summarize the main contributions of this work, discuss the impact that it may have in different research communities, and outline its future directions.

17.1 Conclusions

In this thesis, we mainly focused our attention on the decentralized aspects concerning the runtime verification of distributed intelligent systems (such as MAS); in particular, the verification of interaction protocols (Chapter 8). We showed how these scenarios present peculiar issues which must be seriously addressed, such as the bottleneck problem which is common in the MAS field when the number of interacting agents increases. Decentralizing the runtime verification using multiple monitors is a fine and scalable solution to this problem (Chapter 9).

Since reliability is a key aspect for distributed artificial systems, we also dedicated part of the thesis to studying a probabilistic approach for verifying interaction protocols and showed how to efficiently exploit it in the MAS scenario (Chapter 10), also taking care of the decentralized aspects in the process.

During the 3 years of Ph.D. we have studied and experimented the verification of MAS not only at runtime, but also at static time. In Chapter 12 we presented our approach for applying formal verification, e.g. model checking, directly on top of trace expression specifications. In the thesis we showed how this kind of approach allowed us to make the resulting runtime verification process more reliable. We also experimented the advantages of bringing runtime verification inside model checking (Chapter 13), in particular inside a famous model checker framework for MAS, namely MCAPL. The integration of our specifications into this engine has brought advantages – showed through experimental results – such as the reduction of the size of the model used inside MCAPL, with a consequent smaller number of states to be analyzed (increasing the performances of the static verification process).

The contributions of the Ph.D. more concerned with engineering and practical aspects are presented at the end of the thesis (Chapter 14) and show the tool RIVERtools which has been developed for supporting the specification of trace expressions, alongside syntactic and (some) semantic controls, and tools for both centralized and decentralized runtime verification. The usage of RIVERtools has been shown in connection with the JADE platform although it has been designed to be – in principle – connected to any other framework. Finally, we have presented a case study (Chapter 15) where trace expressions demonstrate all their potential for representing complex protocols, statically checking the properties of these protocols, and creating agents driven by them.

Trace expressions and RV are fascinating and through the works presented in this thesis we showed many possible uses and studied different theoretical and practical aspects. Other colleagues are also planning to extend and use the trace expression formalism. More specifically, Ancona, Franceschini et al. are also considering a more radical extension of the formalism in order to make it more compact, expressive and reusable. In the last years, they have been focusing more on the use of trace expressions in the IoT field (Ancona et al., 2017b; Leotta et al., 2018): they are planning to use them to generate more invasive monitors which are able to intercept and filter events at the edge of

the heterogeneous IoT system. Many applications, such as machine learning ones, may gain from this proactive approach where the monitor could help in having a more reliable training phase, filtering the events that are out of a specific range or that do not respect some time constraints.

In the next sections we are going to present expected and **unexpected** future directions of the thesis.

17.2 *Expected Future Directions*

In the following paragraphs we regrouped the future works of the thesis by argument.

FORMALISM EXTENSIONS. We are planning to further experiment with parametric and probabilistic trace expressions to constitute a library of specifications for the most commonly used interaction protocols, and to individuate recurrent patterns that can be usefully exploited to ease the specification of complex protocols. We also want to provide guidelines and automatic tools to support the developer for the probabilistic extension (as it is already for the parametric one). We are currently working to extend RIVERtools (Chapter 14) towards this direction.

DECENTRALIZED RV. In Chapter 9 we presented the *DecAMon* algorithm for achieving the decentralization of the runtime verification of agent interaction protocols. We left for future developments the investigation of suitable heuristics for boosting the efficiency of *DecAMon* when a MAS partition must be recomputed very often; other interesting research directions include the extension of *DecAMon* to deal with parametric trace expressions (Chapter 6), and its exploitation in other promising application domains, such as ambient intelligence systems (Ancona et al., 2015b) and traffic monitoring. As presented in the chapter, we assumed the use of the RSC model for achieving the communication among the agents. As a future work, we are going to present the differences when we consider other less restrictive models. We expect that everything presented in this chapter will still be valid. The only aspect that has to change is the definition of critical points, since it is the part directly related to the concept of Good protocol. For instance, considering the most asynchronous model, it will not be enough to have an agent in common to satisfy a critical point, but will be necessary to have specific agents in common (in the sequence of two messages we should have the receiver and the sender in common for instance).

From the point of view of the decentralization problem applied when gaps are present, in Chapter 10 we discussed our approach focusing only on the presence of full gaps. We are considering also to extend our implementation to cope with partially instantiated gaps. Another future work will be to consider a threshold in order to cut branches that are unreasonable to maintain, as the pobability to be correct is too low. Fixed a threshold, a monitor will be able to remove all the branches with a joint probability associated with them

lower than the chosen threshold. This will bring the advantage of anticipating the error detection and to prune useless branches related to unreasonable possibilities.

CONFORMANCE. In Chapter 11 we presented a conformance test for the trace expression formalism. W.r.t. the five steps introduced in Section 11.1, we exploited achieved results for steps (1-2), we devoted the entire chapter to step (3), and we demonstrated how to tackle step (5). More sophisticated approaches could be followed for step (5), each with pros and cons. Experimenting some of them, such as introducing a mediator agent between *mas* and *mas′* acting as the *i* interface and generating wrappers for the agents that must substitute other agents, will be explored in the future. Step (4) is an open problem which falls outside the scope of our investigation: in Chapter 11 we do not face the issue of "semantic/pragmatic conformance", but only that of "syntactic conformance". In case some constraints on interactions are know, for example commitments that must be fulfilled and that can drive the choice of the most suitable mapping, we might exploit them. Otherwise, by interpreting messages as words or sentences in some natural language, we might take advantage of semantic techniques similar to those used for matching ontological concepts (Mascardi, Locoro, and Rosso, 2010). Ontology matching could hence be exploited in the global process we have presented, in step (4). We remark that if we knew in advance which are the semantically correct message and agents mappings, we could feed our algorithm with them and use it as a "plain conformance checker" like those mentioned above, rather than a "conformant mappings builder". Even if we knew all the correct mappings in advance, however, we could not run any of the existing conformance checking algorithms on trace expressions, because of their higher expressiveness. Finally, an extremely challenging issue would be to identify mappings where one message used in one MAS corresponds to a *sequence* of messages used in another MAS, and to consider message inputs and outputs as well.

COMBINING STATIC AND RUNTIME VERIFICATION. In Chapter 12 we presented a way to directly model check a trace expression. The next steps will be to improve the search for cycles inside expansive terms and, to study in greater detail the expressivity of trace expressions in order to understand if an hybrid approach is possible, where the LTL properties verified statically can bring us to simplify the trace expression generating consequently a simpler version of the monitor. One possible future work will be achieving the static verification also for a parametric trace expression. The presence of parameters makes not usable the standard automata-based approach. One promising way to solve this problem is through the model checker Cubicle (Conchon et al., 2012) (used for symbolic backward reachability analysis on infinite sets of states).

In Chapter 13 we discussed how to use trace expressions for supporting MCAPL. In the future, we aim to provide arguments (ideally proofs) that the behaviour of the abstract environments generated by the system genuinely

expresses the behaviour specified by the trace expressions. It would also be desirable to express a greater range of constraints in these models – for instance, the constraint that some belief can only occur after some action is taken (e.g., that a car can only reach the speed limit after an acceleration has been performed). Finally, we plan to apply our approach to a real case study. The scenario we have in mind is a cyberphysical system which must demonstrate its dependability in order to be acceptable to society and be trusted by its users. As an example, in a remote patient monitoring system where the program integrates sensory input, formal guarantees should be provided that the system respects given medical guidelines (model checking stage), and a RV stage looking at sensors perceptions should monitor that those guidelines are continuously met.

RIVERTOOLS. The future directions of our IDE will include to implement connectors to other target systems. In the MAS context, we plan to add a connector to Jason. Outside the MAS community, we will explore the possibility of an integration with some general purpose system, as Node.js, which has already been used in fascinating scenarios like the Internet of Things (IoT).

17.3 Unexpected Future Directions

Although the reference domain of this thesis is Artificial Intelligence, particularly Distributed Artificial Intelligence and MAS, the impact of trace expressions and their extensions goes far beyond the MAS domain. More specifically, we see potential, although somewhat visionary, applications in the following areas:

- *Protocol-driven chatbots*: During the Ph.D. I have been working on different projects involving industries where the aim was to develop chatbots. Nowadays, developing these systems is almost standardized following a machine learning approach (see Dialogflow[1], Wit.ai[2], IBM Watson[3], AWS Lex[4], and so on), where the intents (the conversations) and the entities (the objects) are defined by the programmer through examples, and the chatbot is trained with them. In these scenarios it is easy to lose control and, if the chatbot is able to dynamically learn from the users, it is hard to always foresee how the chatbot will react. For this reason protocol-driven agents (Ancona et al., 2015a) might be a more suitable solution: instead of training the chatbot using examples we define its behaviours through protocols. This could be relevant in scenarios where it is important to have full control of the chatbot.
- *RV of chatbots*: The initially mentioned frameworks allow a fast and reusable way to produce complex chatbots with no effort. For this reason

[1] https://dialogflow.com
[2] https://wit.ai
[3] https://www.ibm.com/watson/
[4] https://aws.amazon.com/it/lex/

the protocol-driven approach, even though it increases the reliability, could be a less practical solution for developers who are not familiar with formal specifications. Nevertheless, RV could be a more suitable solution that could help in the development of these systems without influencing their standard creation process. Thanks to RV, we can define the protocols the chatbot must respect and then just validate its runtime behaviour. In this way, we can achieve a fast approach supported by many different frameworks without giving up a reliable and controllable system.

- *Self-* systems*[5]: As we mentioned in this thesis, trace expressions have been successfully exploited for modeling protocol-driven agents (Ancona et al., 2015a). With them we can define and implement self-* systems where the components react and change their behaviour dynamically. Protocol-driven agents are a natural choice for modeling these kind of systems since they also intrinsically change their protocols dynamically. Nevertheless, we might need to check our systems – especially if they are implemented by third parties – and monitors have to present a certain level of adaptability. A suitable and challenging use of the work presented in this thesis could be to model a "self-adaptive" monitor, by statically analysing the trace expression (for instance through model checking, see Chapter 12) the monitor would be able to check if some requirements are met and, if not, communicate with other monitors in order to find more suitable trace expressions. In case the analyzed agents would also be "reflective"[6], adaptivity could take place at the agent level, and not only at the MAS level.
- *IoT*: Trace expressions have already been exploited in IoT scenarios (Ancona et al., 2017b), but only from a centralized point of view. One of the main contributions of this thesis has been studying and defining an approach to decentralize the RV of MAS. IoT systems are as distributed as MAS and can benefit from our approach as well; they usually are heterogeneously distributed and are strongly dependent on the environment they are exploited in. During the Ph.D. we thought many times on how the system's topology could influence the choice of the partitions of monitors (see Chapter 9). In the MAS field this could be relevant, but in many scenarios the agents are used as an abstraction of our real entities and the presence of general and well-supported frameworks (such as Jason and JADE) makes the topology of the system less important[7]. In IoT scenarios instead, all the components are implicitly situated inside a real environment, usually at a lower level. In these scenarios the distribution

[5]Systems satisfying the Self-* properties defined by IBM (Poslad, 2009): Self configuration, Self-healing, Self-optimization, Self-protection.

[6]In computer science, reflection is the ability of a computer program to examine, introspect, and modify its own structure and behavior at runtime (`https://www.researchgate.net/profile/Jacques_Malenfant/publication/243671255_A_Tutorial_on_Behavioral_Reflection_and_its_Implementation`).

[7]This is a conjecture, we should investigate more on this.

of the RV process may be affected by the choice of the partitions and an integration of the work presented in Chapter 9 could be really useful.

- *Mixing RV and testing*: In this thesis we focused on RV and its combination with static verification (such as Model Checking). As we mentioned, there are three main types of verification of a software system: static verification, runtime verification and testing. RV is in a certain way a midway approach between static verification and testing. After having studied the combination between static and runtime verification, it is natural to wonder about the connections between RV and testing. It would be useful to find a way for supporting testing using runtime monitors; more specifically, testing is well-know to be a hard discipline where we try to cover all the behaviours of a software system through groups of tests. A possible work – challenging as it may be – could be integrating RV and automated testing[8]. We could automatically extract tests from a trace expression, generating monitors only for the paths and branches of the system which are not covered: since a part is statically tested, this would simplify trace expressions and make the testing more reliable since the uncovered branches would be handled by the runtime monitors.

- *Team building*: During the Ph.D. program I have also been working on team building projects. In these scenarios the aim is to create serious games which are used to help people to cooperate and work together as a team; it is a famous technique used by industries to improve cooperation among their employees which has also been successfully exploited for educational purposes. These scenarios have something in common: they are distributed (as groups, teams and single players) and have rules. For this reason we started to think about the use of RV for achieving the validation of these rules at real time when the game is played. These scenarios are usually unsupported by computer systems but, with the experience made working on this, we concluded that MAS combined with runtime monitoring could be a suitable choice for a better standardization of the approach.

- *Education*: As for team building, MAS could be a useful and challenging solution for educational purposes as well. To initiate the new generations to computer science, many frameworks and techniques have been proposed[9]; among the most successful ones is Scratch[10] which makes it possible to introduce children and teenagers to the basic concepts of computer science in a fun and interactive way. One issue concerning these approaches is the absence of team work among the participants which brings most kids to think computer science as a "one-man band" discipline, contrary to reality. This impression is due to the implicit cent-

[8] Discipline concerning the automatic generation of tests.

[9] It is enough to think about the big participation to events like "The hour of code" `https://hourofcode.com/it`.

[10] `https://scratch.mit.edu`

ralized implementation of the used frameworks: with Scratch, the kid is alone and has to interact with the tool without the possibility to communicate with the others. Exploiting MAS and their implicit distribution to decentralize frameworks like Scratch could help with giving the right impression about our discipline and open many different scenarios of use and application.

- *Trace expressions Inference*: During the Ph.D. we thought many times about possible uses of Grammar Inference to automatically extract trace expressions directly from the system. Grammar Inference provides a tool for automatic acquisition of syntax; it is defined as the process of learning a target grammar from a set of labeled sentences (Parekh and Honavar, 1998). Inducing, learning or inferring grammars has been studied for decades, but only in recent years has grammatical inference emerged as an independent field (Higuera, 2010). Through the works in this thesis we have studied, implemented and experimented trace expressions from many different viewpoints. Nevertheless, each contribution always starts from a trace expression and never the other way around. It could be extremely interesting and relevant to approach the problem taking a different perspective, namely starting from the traces generated by the system and "synthetizing" the corresponding trace expression; once obtained, the trace expression could be used for all the purposes presented in this thesis.

A Medical Guidelines

In this Appendix we represent all the GL properties presented in Bottrighi et al., 2010 with trace expressions. The GL descriptions, comments and relevance are taken from Bottrighi et al., 2010.

Structural properties

Structural properties concern the existence of the appropriate clinical requirements, and are relevant in order to ensure the appropriate management of any patient.
Example
Verify that the GL contains a run not including any surgical intervention (cholelithiasis GL).

$$\exists_{run}.(run \in [\![\tau]\!])$$

$$\tau = (\neg surgery{:}\tau) \vee \epsilon$$

Comment: The property evaluates to true if there is at least a run in which surgery is never performed.
Relevance: The most frequent treatment of gallstones is the expectant management (asymptomatic gallstones).

Example
Verify that any run contains antibiotic treatment (community acquired pneumonia GL).

$$\forall_{run}.(run \in [\![\tau]\!])$$

$$\tau = (antibiotic_treatment{:}\tau_1) \vee (\neg antibiotic_treatment{:}\tau) \qquad \tau_1 = (any{:}\tau_1) \vee \epsilon$$

Comment: The property evaluates to true if all possible runs contain a state in which an antibiotic treatment is administered.
Relevance: The antibiotic treatment is mandatory in the case of community acquired pneumonia.

Example
Verify that the GL contains a run including thrombolysis (ischemic stroke GL).

$$\exists_{run}.(run \in [\![\tau]\!])$$

$$\tau = (trombolysis{:}\tau_1) \vee (\neg trombolysis{:}\tau) \qquad \tau_1 = (any{:}\tau_1) \vee \epsilon$$

Comment: The property evaluates to true if there is at least a run in which thrombolysis is executed.
Relevance: It is important to perform a thrombolysis in case the patient is eligible and in the hospital is present a stroke unit.

Example
Verify that cholecystectomy is not repeated (cholelithiasis GL).

$$\forall_{run}.(run \in [\![\tau]\!])$$

$$\tau = (cholecystectomy{:}\tau_1) \vee (\neg cholecystectomy{:}\tau) \vee \epsilon$$

$$\tau_1 = (\neg cholecystectomy{:}\tau_1) \vee \epsilon$$

Comment: Once removed, an organ cannot be removed again.
Relevance: The surgical treatment of gallstones in a cholecystectomized patient cannot consist of a new cholecystectomy.

Medical validity properties

Medical validity properties concern both the exclusion of dangerous treatments and the inclusion of the most appropriate treatments for the considered class of patients.
Example
Verify that there is a run in which the acetylsalicilic acid (ASA) is not used (ischemic stroke GL).

$$\exists_{run}.(run \in [\![\tau]\!])$$

$$\tau = (\neg ASA{:}\tau) \vee \epsilon$$

Comment: This is the condition for applying the GL in the case the patient is allergic to ASA.
Relevance: The ASA allergy is potentially life threatening.

Example
Verify that if vital signs are altered, the patient is sent to intensive care unit (ischemic stroke GL).

$$\forall_{run}.(run \in [\![\tau]\!])$$

$$\tau = (vital_signs_altered{:}\tau_1) \vee (\neg vital_signs_altered{:}\tau) \vee \epsilon$$

$$\tau_1 = (sent_to_intensive_care_unit{:}\tau_2) \vee (\neg sent_to_intensive_care_unit{:}\tau_1)$$

$$\tau_2 = (any{:}\tau_2) \vee \epsilon$$

Comment: Monitoring and treatment in an intensive care unit is preferable in this case.
Relevance: The admission of the patient with altered vital signs in an intensive care unit is mandatory

Example
Verify that, in the presence of ureteral lithiasis, there is at least one state in which endoscopic removal is considered (urinary stones LG).

$$\exists_{run}.(run \in [\![\tau]\!])$$

$$\tau = (ureteral_lithiasis_present{:}\tau_1) \vee (\neg ureteral_lithiasis_present{:}\tau) \vee \epsilon$$

$$\tau_1 = (endoscopic_removal{:}\tau_2) \vee (\neg endoscopic_removal{:}\tau_1)$$

$$\tau_2 = (any{:}\tau_2) \vee \epsilon$$

Comment: Alternative treatments (e.g., surgery) are possible, but more invasive.
Relevance: The endoscopic removal of urinary stones, whenever possible, is preferable.

Contextualization properties

contextualization properties lead from (general) GLs to hospital-specific GLs, mainly as concerns the presence/absence of instrumentation.
Example
Verify that each action costs less than X, with X fixed at design time.

$$\forall_{run}.(run \in [\![\tau]\!])$$

$$\tau = cost_of_done(X){:}\tau$$

$$cost_of_done(X)) = \{Action | cost_of(Action) < X\}$$

Comment: Cost reduction is one of the most important aspects of health care policy.

Example
Verify that if the CT scanner is not available, patient transfer to another hospital is considered to perform the test (ischemic stroke GL).

$$\forall_{run}.(run \in [\![\tau]\!])$$

$$\tau = (CT_scanner_absent{:}\tau_1) \lor (\neg CT_scanner_absent{:}\tau) \lor \epsilon$$
$$\tau_1 = (patient_transfer{:}\tau_2) \lor (\neg patient_transfer{:}\tau_1)$$
$$\tau_2 = (any{:}\tau_2) \lor \epsilon$$

Comment: The suspect of ischemic stroke should always be confirmed by CT.
Relevance: The CT scan is almost always diagnostic of the ischemic event and influences the subsequent decisions.

Example
Verify whether Magnetic Resonance (MR) is used in the GL (ischemic stroke GL).

$$\exists_{run}.(run \in [\![\tau]\!])$$

$$\tau = (MR_imaging{:}\tau_1) \lor (\neg MR_imaging{:}\tau)$$
$$\tau_1 = (any{:}\tau_1) \lor \epsilon$$

Comment: This property should be verified since, in the case MR imaging is considered, its execution must be possible.
Relevance: The MR imaging, when available, is the most diagnostic test in some cases of ischemic stroke.

Properties about the application of a GL to a specific patient

These properties are relevant to ensure a patient-centred approach, considering the peculiarity of each patient.
Example
Verify that, after a cerebral hemorrhagic event, no anticoagulant drug is administered (ischemic stroke GL).

$$\forall_{run}.(run \in [\![\tau]\!])$$
$$\tau = (cerebral_hemorrhagic_event{:}\tau_1) \lor \epsilon$$
$$\tau_1 = (\neg anticoagulant_drug_administration{:}\tau_1) \lor \epsilon$$

Comment: In the natural evolution of an ischemic stroke, intra-cranial bleeding may be life-threatening.
Relevance: The intra-cranial hemorrhage is an absolute contraindication to anticoagulant drug administration.

Example
Verify that, if an infection arises, there is a treatment not based on penicillin.

$$\exists_{run}.(run \in [\![\tau]\!])$$

$$\tau = (infection_value_present{:}\tau_1) \lor (\neg infection_value_present{:}\tau) \lor \epsilon$$
$$\tau_1 = (\neg penicillin_administration{:}\tau_1) \lor \epsilon$$

Comment: Notice that, e.g., penicillin treatment cannot be administered to allergic patients.
Relevance: Antibiotic treatment is based on different drugs which should be adapted to the patient's different conditions.

Example
Verify that, if hyperpyrexia appears, hemoculture is performed (FUO GL).

$$\forall_{run}.(run \in [\![\tau]\!])$$

$$\tau = (hyperpyrexia_value_present{:}\tau_1) \lor (\neg hyperpyrexia_value_present{:}\tau) \lor \epsilon$$
$$\tau_1 = (hemoculture{:}\tau) \lor (\neg hemoculture{:}\tau_1)$$

Comment: In each run, hemoculture must be performed, if hyperpyrexia appears.
Relevance: Hemoculture is very important in the diagnostic process of the FUO because it can

positively influence the antibiotic choice.

Example
Verify that there is a treatment in which growth factors are administered, when leukopenia appears (lymphoma treatment GL).

$$\exists_{run}.(run \in [\![\tau]\!])$$

$$\tau = (leukopenia_value_present{:}\tau_1) \vee (\neg leukopenia_value_present{:}\tau) \vee \epsilon$$

$$\tau_1 = (growth_factors_administration{:}\tau) \vee (\neg growth_factors_administration{:}\tau_1)$$

Comment: The growth factors administration can positively reduce the duration of the leukopenia and the risk of infections.
Relevance: If leukopenia is not severe, there are also alternative treatments to the administration of growth factors. That is why we check the existence of one run in which growth factors are administered, without forcing that they are administered in all runs.

Other complex properties

Example
Verify whether the GL can apply to patients having given features (monitored at subsequent states; ischemic stroke GL).

$$\exists_{run}.(run \in [\![\tau]\!])$$

$$\tau = ((dysphagia_value_present{:}\epsilon) |$$
$$(swallowing_test1_value_positive{:}\epsilon) |$$
$$(speech_language_evaluation_positive{:}\epsilon) |$$
$$(videofluorography_value_positive{:}\epsilon)) \cdot \tau_1$$

$$\tau_1 = any{:}\tau_1$$

Comment: This sequential approach is recommended in evaluating dysphagia.
Relevance: A complete evaluation of dysphagia should include all the above aspects sequentially.

Example
Verifying whether the GL always prescribes the above sequence of actions for patients with dysphagia (ischemic stroke GL).

$$\forall_{run}.(run \in [\![\tau]\!])$$

$$\tau = (dysphagia_value_present{:}\tau_1) \vee (\neg dysphagia_value_present{:}\tau) \vee \epsilon$$

$$\tau_1 = (swallowing_test1{:}\tau_2) \vee (swallowing_test_value_positive{:}\tau_3) \vee$$
$$((\neg swallowing_test1{:}\epsilon) \wedge (\neg swallowing_test_value_positive{:}\epsilon) \cdot \tau_1) \vee \epsilon$$

$$\tau_2 = (swallowing_test_value_positive{:}\tau_4) \vee (\neg swallowing_test_value_positive{:}\tau_2) \vee \epsilon$$

$$\tau_3 = (swallowing_test1{:}\tau_4) \vee (\neg swallowing_test1{:}\tau_3) \vee \epsilon$$

$$\tau_4 = (speech_language_test{:}\tau_5) \vee (speech_language_evaluation_positive{:}\tau_6) \vee$$
$$((\neg speech_language_test{:}\epsilon) \wedge (\neg speech_language_evaluation_positive{:}\epsilon) \cdot \tau_4) \vee \epsilon$$

$$\tau_5 = (speech_language_evaluation_positive{:}\tau_7) \vee (\neg speech_language_evaluation_positive{:}\tau_5) \vee \epsilon$$

$$\tau_6 = (speech_language_test{:}\tau_7) \vee (speech_language_test{:}\tau_6) \vee \epsilon$$

$$\tau_7 = (videofluorography{:}\tau_8) \vee (videofluorography_value_positive{:}\tau_9) \vee$$
$$((\neg videofluorography{:}\epsilon) \wedge (\neg videofluorography_value_positive{:}\epsilon) \cdot \tau_7) \vee \epsilon$$

$$\tau_8 = (videofluorography_value_positive{:}\tau_{10}) \vee (\neg videofluorography_value_positive{:}\tau_8) \vee \epsilon$$

$$\tau_9 = (videofluorography{:}\tau_{10}) \vee (\neg videofluorography{:}\tau_9) \vee \epsilon$$

$$\tau_{10} = (artificial_nutrition{:}\tau) \vee (\neg artificial_nutrition{:}\tau_{10})$$

Comment: A correct treatment (artificial nutrition) is mandatory in confirmed dysphagia.

Relevance: To perform a correct artificial nutrition all the above diagnostic aspects of dysphagia must be evaluated.

Bibliography

Ahrendt, Wolfgang, Gordon J. Pace, and Gerardo Schneider (2016). "StaRVOOrS - Episode II - Strengthen and Distribute the Force". In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, pp. 402–415. DOI: 10.1007/978-3-319-47166-2_28. URL: https://doi.org/10.1007/978-3-319-47166-2_28 (cited on pp. 10, 162).

Ahrendt, Wolfgang, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, eds. (16th Dec. 2016). *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6. URL: http://dx.doi.org/10.1007/978-3-319-49812-6. published (cited on pp. 10, 162).

Aielli, Federica, Davide Ancona, Pasquale Caianiello, Stefania Costantini, Giovanni De Gasperis, Antinisca Di Marco, Angelo Ferrando, and Viviana Mascardi (2016). "FRIENDLY & KIND with your Health: Human-Friendly Knowledge-INtensive Dynamic Systems for the e-Health Domain". In: *Proc. of the International Workshops of PAAMS 2016*. Vol. 616. Communications in Computer and Information Science. Springer, pp. 15–26 (cited on p. 126).

Alberti, Marco, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni (2005). "The *S*CIFF Abductive Proof-Procedure". In: *Proc. of AI*IA 2005*. Vol. 3673. LNCS. Springer, pp. 135–147 (cited on pp. 98, 147, 182).

Alberti, Marco, Marco Gavanelli, Evelina Lamma, Federico Chesani, Paola Mello, and Paolo Torroni (2006). "Compliance verification of agent interaction: a logic-based software tool". In: *Applied Artificial Intelligence* 20.2-4, pp. 133–157 (cited on p. 147).

Alechina, Natasha, Mehdi Dastani, and Brian Logan (2014). "Norm approximation for imperfect monitors". In: *Proc. of AAMAS 2014*. IFAAMAS/ACM, pp. 117–124 (cited on p. 100).

Alotaibi, Hind and Hussein Zedan (2010). "Runtime verification of safety properties in multi-agents systems". In: *Proc. of ISDA 2010*. IEEE, pp. 356–362 (cited on p. 196).

Alur, Rajeev, Thomas A. Henzinger, Gerardo Lafferriere, and George J. Pappas (2000). "Discrete Abstractions of Hybrid Systems". In: *Proc. of the IEEE* 88.7, pp. 971–984 (cited on p. 181).

Ancona, D., S. Drossopoulou, and V. Mascardi (2012). "Automatic Generation of Self-Monitoring MASs from Multiparty Global Session Types in Jason". In: *Proc. of DALT 2012*. Vol. 7784. LNAI. Springer, pp. 76–95 (cited on p. 37).

Ancona, D., A. Ferrando, and V. Mascardi (2017). "Parametric Runtime Verification of Multiagent Systems (extended abstract)". In: *Proc. of the 16th*

Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017. Ed. by Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee. ACM, pp. 1457–1459 (cited on pp. 8, 68).

Ancona, Davide, Matteo Barbieri, and Viviana Mascardi (2013). "Constrained global types for dynamic checking of protocol conformance in multi-agent systems". In: *Proc. of the 28th Annual ACM Symposium on Applied Computing, SAC '13*. Ed. by Sung Y. Shin and José Carlos Maldonado. ACM, pp. 1377–1379. DOI: 10.1145/2480362.2480620. URL: http://doi.acm.org/10.1145/2480362.2480620 (cited on pp. 37, 42, 234).

Ancona, Davide, Angelo Ferrando, and Viviana Mascardi (2016). "Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday". In: Cham: Springer International Publishing. Chap. Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification, pp. 47–64. ISBN: 978-3-319-30734-3 (cited on pp. 8, 40, 234).

– (2018a). "Agents Interoperability via Conformance Modulo Mapping". In: *Proceedings of the 19th Workshop "From Objects to Agents", Palermo, Italy, June 28-29, 2018*. Ed. by Massimo Cossentino, Luca Sabatucci, and Valeria Seidita. Vol. 2215. CEUR Workshop Proceedings. CEUR-WS.org, pp. 109–115. URL: http://ceur-ws.org/Vol-2215/paper_18.pdf (cited on pp. 9, 142).

– (2018b). "Improving flexibility and dependability of remote patient monitoring with agent-oriented approaches". In: *Int. J. Agent-Oriented Software Engineering* 6, Nos. 3/4 (cited on pp. 11, 58, 219).

Ancona, Davide, Daniela Briola, Amal El Fallah Seghrouchni, Viviana Mascardi, and Patrick Taillibert (2014). "Efficient Verification of MASs with Projections". In: *Engineering Multi-Agent Systems - Second International Workshop, EMAS 2014, Revised Selected Papers*. Vol. 8758. LNCS, pp. 246–270 (cited on pp. 99, 100, 234).

Ancona, Davide, Daniela Briola, Angelo Ferrando, and Viviana Mascardi (2015a). "Global Protocols as First Class Entities for Self-Adaptive Agents". In: *Proc. of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015*. Ed. by Gerhard Weiss, Pinar Yolum, Rafael H. Bordini, and Edith Elkind. ACM, pp. 1019–1029. URL: http://dl.acm.org/citation.cfm?id=2773282 (cited on pp. 42, 59, 98, 219, 221, 228, 234, 235, 241, 242).

– (2015b). "Runtime verification of fail-uncontrolled and ambient intelligence systems: A uniform approach". In: *Intelligenza Artificiale* 9.2, pp. 131–148. DOI: 10.3233/IA-150084. URL: http://dx.doi.org/10.3233/IA-150084 (cited on pp. 234, 239).

– (2016). "MAS-DRiVe: a Practical Approach to Decentralized Runtime Verification of Agent Interaction Protocols". In: *Proc. of the 17th Workshop "From Objects to Agents"*. Ed. by Corrado Santoro, Fabrizio Messina, and Massimiliano De Benedetti. Vol. 1664. CEUR Workshop Proceedings. CEUR-WS.org, pp. 35–43. URL: http://ceur-ws.org/Vol-1664/w7.pdf (cited on pp. 9, 99, 100, 234).

Ancona, Davide, Angelo Ferrando, Luca Franceschini, and Viviana Mascardi (2017a). "Parametric Trace Expressions for Runtime Verification of Java-Like Programs". In: *Proc. of the 19th Workshop on Formal Techniques for Java-like Programs.* ACM, 10:1–10:6. DOI: 10.1145/3103111.3104037. URL: http://doi.acm.org/10.1145/3103111.3104037 (cited on pp. 8, 180, 234, 236).

Ancona, Davide, Luca Franceschini, Giorgio Delzanno, Maurizio Leotta, Marina Ribaudo, and Filippo Ricca (2017b). "Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things". In: *ALP4IoT@iFM.* Vol. 264. EPTCS, pp. 27–42 (cited on pp. 180, 236, 238, 242).

Ancona, Davide, Angelo Ferrando, Luca Franceschini, and Viviana Mascardi (2018a). "Coping with Bad Agent Interaction Protocols When Monitoring Partially Observable Multiagent Systems". In: *PAAMS.* Vol. 10978. Lecture Notes in Computer Science. Springer, pp. 59–71 (cited on pp. 9, 97).

– (2018b). "Managing Bad AIPs with RIVERtools". In: *PAAMS.* Vol. 10978. Lecture Notes in Computer Science. Springer, pp. 296–300 (cited on pp. 10, 195).

Arrott, Matthew, Alan D Chave, Claudiu Farcas, Emilia Farcas, Jack E Kleinert, Ingolf Krueger, Michael Meisinger, John A Orcutt, Cheryl Peach, Oscar Schofield, et al. (2009). "Integrating marine observatories into a system-of-systems: Messaging in the US Ocean Observatories Initiative". In: *OCEANS 2009, MTS/IEEE Biloxi-Marine Technology for Our Future: Global and Local Challenges.* IEEE, pp. 1–9 (cited on p. 63).

Artho, Cyrille and Armin Biere (2005). "Combined Static and Dynamic Analysis". In: *Electr. Notes Theor. Comput. Sci.* 131, pp. 3–14. DOI: 10.1016/j.entcs.2005.01.018. URL: https://doi.org/10.1016/j.entcs.2005.01.018 (cited on pp. 10, 162).

Artho, Cyrille, Viktor Schuppan, Armin Biere, Pascal Eugster, Marcel Baur, and Boris Zweimüller (2004). "JNuke: Efficient Dynamic Analysis for Java". In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pp. 462–465. DOI: 10.1007/978-3-540-27813-9_37. URL: https://doi.org/10.1007/978-3-540-27813-9_37 (cited on pp. 10, 162).

Austin, J.L. (1962). *How to Do Things with Words.* Oxford (cited on p. 18).

Avizienis, Algirdas (1967). "Design of Fault-tolerant Computers". In: *Proc. of the American Federation of Information Processing Societies Joint Computer Conference, AFIPS '67 (Fall).* New York, NY, USA: ACM, pp. 733–743. DOI: 10.1145/1465611.1465708. URL: http://doi.acm.org/10.1145/1465611.1465708 (cited on p. 59).

Avizienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr (Jan. 2004). "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Trans. Dependable Secur. Comput.* 1.1, pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2. URL: http://dx.doi.org/10.1109/TDSC.2004.2 (cited on pp. 59, 221).

Babaee, Reza, Arie Gurfinkel, and Sebastian Fischmeister (2018). "*P*revent : A Predictive Run-Time Verification Framework Using Statistical Learning". In: *SEFM*. Vol. 10886. Lecture Notes in Computer Science. Springer, pp. 205–220 (cited on p. 80).

Bakar, Najwa Abu and Ali Selamat (2013). "Runtime Verification of Multi-agent Systems Interaction Quality". In: *Proc. of ACIIDS 2013*, pp. 435–444. DOI: 10.1007/978-3-642-36546-1_45. URL: https://doi.org/10.1007/978-3-642-36546-1_45 (cited on p. 196).

Baldoni, M., C. Baroglio, A. Martelli, and V. Patti (2005a). "Verification of Protocol Conformance and Agent Interoperability". In: *Proc. of CLMAS 2005, Revised Selected and Invited Papers*. Vol. 3900. LNCS. Springer, pp. 265–283 (cited on pp. 98, 146).

Baldoni, Matteo, Cristina Baroglio, and Federico Capuzzimati (2013). "2COMM: A Commitment-Based MAS Architecture". In: *EMAS@AAMAS*. Vol. 8245. LNCS, pp. 38–57 (cited on p. 146).

– (2014a). "A Commitment-Based Infrastructure for Programming Socio-Technical Systems". In: *ACM Trans. Internet Techn.* 14.4, 23:1–23:23. DOI: 10.1145/2677206. URL: http://doi.acm.org/10.1145/2677206 (cited on p. 129).

– (2014b). "Typing Multi-Agent Systems via Commitments". In: *EMAS@AAMAS*. Vol. 8758. LNCS, pp. 388–405 (cited on p. 146).

Baldoni, Matteo, Cristina Baroglio, and Elisa Marengo (2010). "Behavior-Oriented Commitment-based Protocols". In: *Proc. of ECAI 2010 - 19th European Conference on Artificial Intelligence*. Ed. by Helder Coelho, Rudi Studer, and Michael Wooldridge. Vol. 215. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 137–142. URL: http://www.booksonline.iospress.nl/Content/View.aspx?piid=17729 (cited on p. 62).

Baldoni, Matteo, Cristina Baroglio, Alberto Martelli, Viviana Patti, and Claudio Schifanella (2004). "Verifying Protocol Conformance for Logic-Based Communicating Agents". In: *CLIMA*. Vol. 3487. LNCS, pp. 196–212 (cited on p. 146).

– (2005b). "Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step". In: *EPEW/WS-FM*. Vol. 3670. LNCS, pp. 257–271 (cited on pp. 146, 152).

Baldoni, Matteo, Cristina Baroglio, Alberto Martelli, and Viviana Patti (2006). "A Priori Conformance Verification for Guaranteeing Interoperability in Open Environments". In: *ICSOC*. Vol. 4294. LNCS, pp. 339–351 (cited on pp. 146, 152).

Baldoni, Matteo, Cristina Baroglio, Amit K. Chopra, Nirmit Desai, Viviana Patti, and Munindar P. Singh (2009). "Choice, interoperability, and conformance in interaction protocols and service choreographies". In: *AAMAS (2)*. IFAAMAS, pp. 843–850 (cited on p. 146).

Baldoni, Matteo, Cristina Baroglio, Elisa Marengo, Viviana Patti, and Federico Capuzzimati (2014). "Engineering commitment-based business protocols with the 2CL methodology". In: *Autonomous Agents and Multi-Agent Systems* 28.4, pp. 519–557 (cited on p. 63).

Baldoni, Matteo, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio (2015). "Exploiting Social Commitments in Programming Agent Interaction". In: *Proc. of PRIMA 2015*. Vol. 9387. LNCS. Springer, pp. 566–574 (cited on p. 129).

– (2018). "Type checking for protocol role enactments via commitments". In: *Autonomous Agents and Multi-Agent Systems* 32.3, pp. 349–386. DOI: `10.1007/s10458-018-9382-3`. URL: `https://doi.org/10.1007/s10458-018-9382-3` (cited on p. 146).

Barringer, Howard, Alex Groce, Klaus Havelund, and Margaret H. Smith (2010). "Formal Analysis of Log Files". In: *JACIC* 7.11, pp. 365–390 (cited on p. 81).

Bartocci, Ezio (2013). "Sampling-based Decentralized Monitoring for Networked Embedded Systems". In: *HAS*. Vol. 124. EPTCS, pp. 85–99 (cited on p. 128).

Bartocci, Ezio and Yliès Falcone, eds. (2018). *Lectures on Runtime Verification - Introductory and Advanced Topics*. Vol. 10457. Lecture Notes in Computer Science. Springer (cited on p. 80).

Bartocci, Ezio, Radu Grosu, Panagiotis Katsaros, C. R. Ramakrishnan, and Scott A. Smolka (2011). "Model Repair for Probabilistic Systems". In: *TACAS*. Vol. 6605. Lecture Notes in Computer Science. Springer, pp. 326–340 (cited on p. 80).

Bartocci, Ezio, Yliès Falcone, Adrian Francalanza, and Giles Reger (2018). "Introduction to Runtime Verification". In: *Lectures on Runtime Verification*. Vol. 10457. Lecture Notes in Computer Science. Springer, pp. 1–33 (cited on p. 80).

Basin, David A., Felix Klaedtke, Srdjan Marinovic, and Eugen Zalinescu (2012). "Monitoring Compliance Policies over Incomplete and Disagreeing Logs". In: *Proc. of RV 2012, Revised Selected Papers*. Vol. 7687. LNCS. Springer, pp. 151–167 (cited on p. 101).

Bauer, Andreas, Martin Leucker, and Christian Schallhart (2009). "Runtime Verification for LTL and TLTL". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)*. in press (cited on pp. 32, 33, 41, 162).

Bauer, Bernhard, Jörg P Müller, and James Odell (2001). "Agent UML: A formalism for specifying multiagent software systems". In: *International journal of software engineering and knowledge engineering* 11.03, pp. 207–230 (cited on p. 60).

Baum, Leonard E. and Ted Petrie (Dec. 1966). "Statistical Inference for Probabilistic Functions of Finite State Markov Chains". In: *Ann. Math. Statist.* 37.6, pp. 1554–1563. DOI: `10.1214/aoms/1177699147`. URL: `https://doi.org/10.1214/aoms/1177699147` (cited on p. 36).

Bayliss, Elizabeth A., John F. Steiner, Douglas H. Fernald, Lori A. Crane, and Deborah S. Main (2003). "Descriptions of barriers to self-care by persons with comorbid chronic diseases". In: *Annals of Family Medicine* 1.1, pp. 15–21 (cited on p. 220).

Bellifemine, Fabio Luigi, Giovanni Caire, and Dominic Greenwood (2007). *Developing Multi-Agent Systems with JADE*. Wiley (cited on pp. 10, 21, 146, 235).

Benerecetti, Massimo, Fausto Giunchiglia, and Luciano Serafini (1998). "Model Checking Multiagent Systems". In: *J. Log. Comput.* 8.3, pp. 401–423. DOI: 10.1093/logcom/8.3.401. URL: https://doi.org/10.1093/logcom/8.3.401 (cited on p. 181).

Bertot, Yves and Pierre Castran (2010). *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. Springer Publishing Company, Incorporated. ISBN: 3642058809, 9783642058806 (cited on p. 30).

Bettini, Lorenzo, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida (2008). "Global Progress in Dynamically Interleaved Multiparty Sessions". In: *Proc. of CONCUR 2008*. Vol. 5201. Springer, pp. 418–433 (cited on p. 98).

Bodden, Eric (Mar. 2005). "Efficient and Expressive Runtime Verification for Java". In: *Grand Finals of the ACM Student Research Competition 2005*. URL: http://www.bodden.de/pubs/bodden05efficient.pdf (cited on p. 161).

Boer, Frank S. de, Koen V. Hindriks, Wiebe van der Hoek, and John-Jules Ch. Meyer (2007). "A verification framework for agent programming with declarative goals". In: *J. Applied Logic* 5.2, pp. 277–302. DOI: 10.1016/j.jal.2005.12.014. URL: https://doi.org/10.1016/j.jal.2005.12.014 (cited on p. 21).

Bonakdarpour, Borzoo, Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers, eds. (2016a). *Bertinoro Seminar on Distributed Runtime Verification, May 2016, Available from http://www.labri.fr/perso/travers/DRV2016/* (cited on pp. 80, 128).

– (2016b). "Challenges in Fault-Tolerant Distributed Runtime Verification". In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016. Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Springer, pp. 363–370. ISBN: 978-3-319-47169-3. DOI: 10.1007/978-3-319-47169-3_27. URL: http://dx.doi.org/10.1007/978-3-319-47169-3_27 (cited on pp. 111, 128).

Bonnell, Sarah and Suneet Mittal (2013). "Clinical Guidelines for Remote Monitoring". In: *Cardiac Electrophysiology Clinics* 5.3. Remote Monitoring and Physiologic Sensing Technologies, pp. 283 –291. ISSN: 1877-9182. DOI: https://doi.org/10.1016/j.ccep.2013.05.003. URL: http://www.sciencedirect.com/science/article/pii/S1877918213000610 (cited on p. 222).

Bordeaux, Lucas, Gwen Salaün, Daniela Berardi, and Massimo Mecella (2004). "When are Two Web Services Compatible?" In: *TES*. Vol. 3324. LNCS, pp. 15–28 (cited on p. 147).

Bordini, R. H., J. F. Hübner, and M. Wooldridge (2007). *Programming Multi-Agent Systems in AgentSpeak Using Jason.* John Wiley & Sons (cited on pp. 20, 235).

Bordini, Rafael H., Michael Fisher, Willem Visser, and Michael Wooldridge (2004). "Model Checking Rational Agents". In: *IEEE Intelligent Systems* 19.5, pp. 46–52. DOI: 10.1109/MIS.2004.47. URL: https://doi.org/10.1109/MIS.2004.47 (cited on p. 35).

– (2006). "Verifying Multi-agent Programs by Model Checking". In: *Autonomous Agents and Multi-Agent Systems* 12.2, pp. 239–256. DOI: 10.1007/s10458-006-5955-7. URL: https://doi.org/10.1007/s10458-006-5955-7 (cited on pp. 3, 35, 182).

Bordini, Rafael H., Louise A. Dennis, Berndt Farwer, and Michael Fisher (2008). "Automated Verification of Multi-Agent Programs". In: *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy.* IEEE Computer Society, pp. 69–78. DOI: 10.1109/ASE.2008.17. URL: https://doi.org/10.1109/ASE.2008.17 (cited on p. 36).

Borst, Willem Nico (1997). "Construction of Engineering Ontologies for Knowledge Sharing and Reuse." base-search.net (ftunivtwente:oai:doc.utwente.nl:17864). PhD thesis. University of Twente, Enschede, Netherlands (cited on p. 19).

Bottrighi, Alessio, Laura Giordano, Gianpaolo Molino, Stefania Montani, Paolo Terenziani, and Mauro Torchio (2010). "Adopting model checking techniques for clinical guidelines verification". In: *Artificial Intelligence in Medicine* 48.1, pp. 1–19. DOI: 10.1016/j.artmed.2009.09.003. URL: https://doi.org/10.1016/j.artmed.2009.09.003 (cited on pp. 221–223, 228, 245).

Bratman, M. (1987). *Intention, plans, and practical reason.* Cambridge, MA: Harvard University Press. ISBN: 9780674458185. URL: http://books.google.de/books?id=I0nuAAAAMAAJ (cited on p. 20).

Bratman, Michael E., David J. Israel, and Martha E. Pollack (1988). "Plans and resource-bounded practical reasoning". In: *Computational Intelligence* 4.3, pp. 349–355. ISSN: 1467-8640. DOI: 10.1111/j.1467-8640.1988.tb00284.x. URL: http://dx.doi.org/10.1111/j.1467-8640.1988.tb00284.x (cited on p. 20).

Bravetti, Mario and Gianluigi Zavattaro (2007a). "A Theory for Strong Service Compliance". In: *COORDINATION.* Vol. 4467. LNCS, pp. 96–112 (cited on p. 147).

– (2007b). "Contract Based Multi-party Service Composition". In: *FSEN.* Vol. 4767. LNCS, pp. 207–222 (cited on p. 147).

Briola, Daniela and Viviana Mascardi (2011). "Design and Implementation of a NetLogo Interface for the Stand-Alone FYPA System". In: *Proce. of WOA, the 12th Workshop on Objects and Agents.* Ed. by Giancarlo Fortino, Alfredo Garro, Luigi Palopoli, Wilma Russo, and Giandomenico Spezzano. Vol. 741. CEUR Workshop Proceedings. CEUR-WS.org, pp. 41–50. URL: http://ceur-ws.org/Vol-741/ID10_Briola_Mascardi.pdf (cited on p. 235).

Briola, Daniela, Viviana Mascardi, and Davide Ancona (2014a). "Distributed Runtime Verification of JADE Multiagent Systems". In: *Proc. of the 8th International Symposium on Intelligent Distributed Computing, IDC 2014*. Ed. by David Camacho, Lars Braubach, Salvatore Venticinque, and Costin Badica. Vol. 570. Studies in Computational Intelligence. Springer, pp. 81–91. DOI: 10.1007/978-3-319-10422-5_10. URL: http://dx.doi.org/10.1007/978-3-319-10422-5_10 (cited on p. 234).

– (2014b). "Distributed Runtime Verification of JADE and Jason Multiagent Systems with Prolog". In: *Proc. of CILC 2014, the 29th Italian Conference on Computational Logic*. Ed. by Laura Giordano, Valentina Gliozzi, and Gian Luca Pozzato. Vol. 1195. CEUR Workshop Proceedings. CEUR-WS.org, pp. 319–323. URL: http://ceur-ws.org/Vol-1195/short3.pdf (cited on p. 234).

Broy, Manfred, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner (2005a). *Model-Based Testing of Reactive Systems: Advanced Lectures (LNCS)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN: 3540262784 (cited on p. 30).

Broy, Manfred, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, eds. (2005b). *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*. Vol. 3472. Lecture Notes in Computer Science. Springer. ISBN: 3-540-26278-4. DOI: 10.1007/b137241. URL: https://doi.org/10.1007/b137241 (cited on p. 160).

Büchi, J. Richard (1990). "On a Decision Method in Restricted Second Order Arithmetic". In: *The Collected Works of J. Richard Büchi*. Ed. by Saunders Mac Lane and Dirk Siefkes. New York, NY: Springer New York, pp. 425–435. ISBN: 978-1-4613-8928-6. DOI: 10.1007/978-1-4613-8928-6_23. URL: https://doi.org/10.1007/978-1-4613-8928-6_23 (cited on p. 32).

Bulling, Nils, Mehdi Dastani, and Max Knobbout (2013). "Monitoring norm violations in multi-agent systems". In: *Proc. of AAMAS 2013*. IFAAMAS, pp. 491–498 (cited on p. 100).

Busi, Nadia, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro (2005). "Choreography and Orchestration: A Synergic Approach for System Design". In: *ICSOC*. Vol. 3826. LNCS, pp. 228–240 (cited on p. 147).

Calinescu, Radu, Danny Weyns, Simos Gerasimou, Muhammad Usman Iftikhar, Ibrahim Habli, and Tim Kelly (2018). "Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases". In: *IEEE Trans. Software Eng.* 44.11, pp. 1039–1069 (cited on p. 65).

Capecchi, Sara, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, and Elena Giachino (Feb. 2009). "Amalgamating Sessions and Methods in Object-oriented Languages with Generics". In: *Theor. Comput. Sci.* 410.2-3, pp. 142–167. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2008.09.016. URL: http://dx.doi.org/10.1016/j.tcs.2008.09.016 (cited on p. 162).

Casella, Giovanni and Viviana Mascardi (2007). "West2East: exploiting WEb Service Technologies to Engineer Agent-based SofTware". In: *IJAOSE* 1.3/4,

pp. 396–434. DOI: 10.1504/IJAOSE.2007.016267. URL: https://doi.org/10.1504/IJAOSE.2007.016267 (cited on p. 98).

Castagna, Giuseppe, Mariangiola Dezani-Ciancaglini, and Luca Padovani (2012). "On Global Types and Multi-Party Session". In: *Logical Methods in Computer Science* 8.1. DOI: 10.2168/LMCS-8(1:24)2012. URL: https://doi.org/10.2168/LMCS-8(1:24)2012 (cited on pp. 102, 103).

Chaudhry, Sarwat I., Jennifer A. Mattera, Jeptha P. Curtis, John A. Spertus, Jeph Herrin, Zhenqiu Lin, Christopher O. Phillips, Beth V. Hodshon, Lawton S. Cooper, and Harlan M. Krumholz (2010). "Telemonitoring in Patients with Heart Failure". In: *New England Journal of Medicine* 363.24, pp. 2301–2309 (cited on p. 220).

Chen, Bihuan, Xin Peng, Yang Liu, Songzheng Song, Jiahuan Zheng, and Wenyun Zhao (2017). "Architecture-Based Behavioral Adaptation with Generated Alternatives and Relaxed Constraints". In: *IEEE Transactions on Services Computing* PP.99, pp. 1–1 (cited on p. 59).

Chen, F. and G. Rosu (2007). "Mop: an efficient and generic runtime verification framework". In: *OOPSLA 2007*, pp. 569–588 (cited on p. 31).

Chesani, Federico, Paola Mello, Marco Montali, and Paolo Torroni (2009). "Commitment Tracking via the Reactive Event Calculus". In: *Proc. of the 21st International Joint Conference on Artifical Intelligence.* IJCAI'09, pp. 91–96 (cited on pp. 182, 196).

– (2013). "Representing and monitoring social commitments using the event calculus". In: *Autonomous Agents and Multi-Agent Systems* 27.1, pp. 85–130 (cited on pp. 62, 63).

Chevrou, Florent, Aurélie Hurault, and Philippe Quéinnec (2016). "On the diversity of asynchronous communication". In: *Formal Asp. Comput.* 28.5, pp. 847–879. DOI: 10.1007/s00165-016-0379-x. URL: https://doi.org/10.1007/s00165-016-0379-x (cited on p. 108).

Chimento, Jesús Mauricio, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider (2015). "StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java". In: *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pp. 297–305. DOI: 10.1007/978-3-319-23820-3_21. URL: https://doi.org/10.1007/978-3-319-23820-3_21 (cited on pp. 10, 162).

Chopra, Amit K, Samuel H Christie V, and Munindar P Singh (2017). "Splee: a declarative information-based language for multiagent interaction protocols". In: *Proc. of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017.* International Foundation for Autonomous Agents and Multiagent Systems, pp. 1054–1063 (cited on p. 61).

Chopra, Amit K., Samuel Christie, and Munindar P. Singh (2017). "Splee: A Declarative Information-Based Language for Multiagent Interaction Protocols". In: *Proc. of AAMAS 2017.* ACM, pp. 1054–1063 (cited on p. 129).

Chopra, Amit K. and Munindar P. Singh (2006). "Producing Compliant Interactions: Conformance, Coverage, and Interoperability". In: *DALT.* Vol. 4327. LNCS, pp. 1–15 (cited on p. 147).

Chopra, Amit K. and Munindar P. Singh (2007). "Interoperation in Protocol Enactment". In: *DALT*. Vol. 4897. LNCS, pp. 36–49 (cited on p. 147).

– (2015a). "Cupid: Commitments in Relational Algebra". In: *Proc. of the Twenty-Ninth AAAI Conference on Artificial Intelligence.* Ed. by Blai Bonet and Sven Koenig. AAAI Press, pp. 2052–2059. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9938 (cited on pp. 63, 129).

– (2015b). "Generalized Commitment Alignment". In: *Proc. of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015*. Ed. by Gerhard Weiss, Pinar Yolum, Rafael H. Bordini, and Edith Elkind. ACM, pp. 453–461. URL: http://dl.acm.org/citation.cfm?id=2772938 (cited on p. 61).

– (2016). "Custard: Computing Norm States over Information Stores". In: *Proc. of the 2016 International Conference on Autonomous Agents & Multiagent Systems.* Ed. by Catholijn M. Jonker, Stacy Marsella, John Thangarajah, and Karl Tuyls. ACM, pp. 1096–1105. URL: http://dl.acm.org/citation.cfm?id=2937085 (cited on p. 63).

Chow, Tsun S. (1978). "Testing Software Design Modeled by Finite-State Machines". In: *IEEE Trans. Software Eng.* 4.3, pp. 178–187. DOI: 10.1109/TSE.1978.231496. URL: https://doi.org/10.1109/TSE.1978.231496 (cited on p. 160).

Clarke, Edmund M. (2008). "25 Years of Model Checking". In: ed. by Orna Grumberg and Helmut Veith. Berlin, Heidelberg: Springer-Verlag. Chap. The Birth of Model Checking, pp. 1–26. ISBN: 978-3-540-69849-4. DOI: 10.1007/978-3-540-69850-0_1. URL: http://dx.doi.org/10.1007/978-3-540-69850-0_1 (cited on p. 29).

Clarke, Edmund M., Orna Grumberg, and Doron A. Peled (2001). *Model checking*. MIT Press. ISBN: 978-0-262-03270-4. URL: http://books.google.de/books?id=Nmc4wEaLXFEC (cited on p. 160).

Clarke Jr., Edmund M., Orna Grumberg, and Doron A. Peled (1999). *Model Checking*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-03270-8 (cited on pp. 3, 28, 30).

Cohen, J., D. Perrin, and J.-E Pin (1993). "On the expressive power of temporal logic". In: *J.Comput. Syst.Sci. 46*, pp. 271–294 (cited on pp. 32, 57).

Cohen, Philip R. and Hector J. Levesque (1990). "Intention is Choice with Commitment". In: *Artif. Intell.* 42.2-3, pp. 213–261. DOI: 10.1016/0004-3702(90)90055-5. URL: https://doi.org/10.1016/0004-3702(90)90055-5 (cited on p. 20).

Colombo, Christian, Gordon J. Pace, and Gerardo Schneider (2009). "LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)". In: *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pp. 33–37. DOI: 10.1109/SEFM.2009.13. URL: https://doi.org/10.1109/SEFM.2009.13 (cited on pp. 10, 162).

Conchon, Sylvain, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi (2012). "Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems - Tool Paper". In: *CAV*, pp. 718–724 (cited on p. 240).

Cossentino, Massimo (2005). "From requirements to code with the PASSI methodology". In: *Agent-oriented methodologies* 3690, pp. 79–106 (cited on p. 98).

Courcelle, B. (1983). "Fundamental properties of infinite trees". In: 25, pp. 95–169 (cited on p. 43).

Criado, Natalia and Jose M. Such (2017). "Norm Monitoring Under Partial Action Observability". In: *IEEE Trans. Cybernetics* 47.2, pp. 270–282. DOI: 10.1109/TCYB.2015.2513430. URL: https://doi.org/10.1109/TCYB.2015.2513430 (cited on pp. 100, 101).

Dalpiaz, Fabiano, Amit K. Chopra, Paolo Giorgini, and John Mylopoulos (2010). "Adaptation in Open Systems: Giving Interaction Its Rightful Place". In: *Proc. of Conceptual Modeling - ER 2010, 29th International Conference on Conceptual Modeling*. Ed. by Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson C. Woo, and Yair Wand. Vol. 6412. LNCS. Springer, pp. 31–45. DOI: 10.1007/978-3-642-16373-9_3. URL: https://doi.org/10.1007/978-3-642-16373-9_3 (cited on pp. 59, 63).

Dastani, Mehdi, M. Birna van Riemsdijk, and John-Jules Ch. Meyer (2005). "Programming Multi-Agent Systems in 3APL". In: *Multi-Agent Programming: Languages, Platforms and Applications*. Ed. by Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni. Vol. 15. Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, pp. 39–67 (cited on p. 21).

David, Alexandre, Kim G. Larsen, Axel Legay, Marius Mikuăionis, and Danny BØgsted Poulsen (Aug. 2015). "Uppaal SMC Tutorial". In: *Int. J. Softw. Tools Technol. Transf.* 17.4, pp. 397–415. ISSN: 1433-2779. DOI: 10.1007/s10009-014-0361-y. URL: http://dx.doi.org/10.1007/s10009-014-0361-y (cited on p. 65).

Delgado, Nelly, Ann Quiroz Gates, and Steve Roach (Dec. 2004). "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools". In: *IEEE Trans. Softw. Eng.* 30.12, pp. 859–872. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.91. URL: http://dx.doi.org/10.1109/TSE.2004.91 (cited on pp. 3, 29).

Deniélou, P.-M. and N. Yoshida (2012). "Multiparty Session Types Meet Communicating Automata". In: *ESOP'12 (part of ETAPS 2012)*. LNCS. Springer (cited on pp. 51, 123).

Deniélou, Pierre-Malo and Nobuko Yoshida (2012). "Multiparty Session Types Meet Communicating Automata". In: *Proc. of ESOP 2012*. Vol. 7211. LNCS. Springer, pp. 194–213 (cited on pp. 98, 107).

Dennis, L. A., M. Fisher, N. Lincoln, A. Lisitsa, and S. M. Veres (2010). "Declarative Abstractions for Agent Based Hybrid Control Systems". In: *Proc. 8th Int. Workshop on Declarative Agent Languages and Technologies (DALT)*, pp. 96–111 (cited on p. 182).

Dennis, Louise A. (2017). *Gwendolen Semantics: 2017*. Tech. rep. ULCS-17-001. University of Liverpool, Department of Computer Science (cited on pp. 182, 183).

Dennis, Louise A (2018). "The MCAPL Framework including the Agent Infrastructure Layer and Agent Java Pathfinder". In: *The Journal of Open Source Software* 3.24 (cited on p. 180).

Dennis, Louise A. and Berndt Farwer (2008). "Gwendolen: A BDI Language for Verifiable Agents". In: *University of Aberdeen* (cited on p. 21).

Dennis, Louise A., Michael Fisher, Matthew P. Webster, and Rafael H. Bordini (2012). "Model checking agent programming languages". In: *Autom. Softw. Eng.* 19.1, pp. 5–63. DOI: `10.1007/s10515-011-0088-x`. URL: `https://doi.org/10.1007/s10515-011-0088-x` (cited on pp. 35, 180, 182).

Dennis, Louise A., Michael Fisher, Nicholas K. Lincoln, Alexei Lisitsa, and Sandor M. Veres (2014). "Practical verification of decision-making in agent-based autonomous systems". English. In: *Automated Software Engineering*, pp. 1–55. ISSN: 0928-8910. DOI: `10.1007/s10515-014-0168-9`. URL: `http://dx.doi.org/10.1007/s10515-014-0168-9` (cited on pp. 179, 182).

Dennis, Louise A. et al. (2016). "Agent-Based Autonomous Systems and Abstraction Engines: Theory Meets Practice". In: *Towards Autonomous Robotic Systems*. Ed. by Lyuba Alboul, Dana Damian, and Jonathan M. Aitken. Cham: Springer International Publishing, pp. 75–86. ISBN: 978-3-319-40379-3 (cited on pp. 182, 184).

Desai, Ankush, Tommaso Dreossi, and Sanjit A. Seshia (2017). "Combining Model Checking and Runtime Verification for Safe Robotics". In: *RV*. Vol. 10548. Lecture Notes in Computer Science. Springer, pp. 172–189 (cited on p. 181).

Desai, Ankush, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey (2013). "P: safe asynchronous event-driven programming". In: *PLDI*. ACM, pp. 321–332 (cited on p. 181).

Desai, N., A. U. Mallya, A. K. Chopra, and M. P. Singh (2005a). "Interaction protocols as design abstractions for business processes". In: *IEEE Transactions on Software Engineering* 31.12, pp. 1015–1027 (cited on p. 63).

Desai, Nirmit, Amit K. Chopra, and Munindar P. Singh (Oct. 2009). "Amoeba: A Methodology for Modeling and Evolving Cross-organizational Business Processes". In: *ACM Trans. Softw. Eng. Methodol.* 19.2, 6:1–6:45. ISSN: 1049-331X. DOI: `10.1145/1571629.1571632`. URL: `http://doi.acm.org/10.1145/1571629.1571632` (cited on p. 63).

Desai, Nirmit and Munindar P. Singh (2008). "On the Enactability of Business Protocols". In: *Proc. of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008*. Ed. by Dieter Fox and Carla P. Gomes. AAAI Press, pp. 1126–1131. URL: `http://www.aaai.org/Library/AAAI/2008/aaai08-178.php` (cited on pp. 63, 102).

Desai, Nirmit, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh (2005b). "Interaction Protocols as Design Abstractions for Business Processes". In: *IEEE Trans. Software Eng.* 31.12, pp. 1015–1027. DOI: `10.1109/`

TSE.2005.140. URL: https://doi.org/10.1109/TSE.2005.140 (cited on p. 98).

Desai, Nirmit, Amit K. Chopra, Matthew Arrott, Bill Specht, and Munindar P. Singh (2007a). "Engineering Foreign Exchange Processes via Commitment Protocols". In: *Proc. of the 2007 IEEE International Conference on Services Computing (SCC 2007)*. IEEE Computer Society, pp. 514–521. DOI: 10.1109/SCC.2007.58. URL: https://doi.org/10.1109/SCC.2007.58 (cited on p. 63).

Desai, Nirmit, Zhengang Cheng, Amit K. Chopra, and Munindar P. Singh (2007b). "Toward Verification of Commitment Protocols and Their Compositions". In: *Proc. of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS '07. Honolulu, Hawaii: ACM, 33:1–33:3. ISBN: 978-81-904262-7-5. DOI: 10.1145/1329125.1329165. URL: http://doi.acm.org/10.1145/1329125.1329165 (cited on p. 63).

Dezani-Ciancaglini, Mariangiola, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou (2005). "Trustworthy Global Computing: International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005. Revised Selected Papers". In: Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. A Distributed Object-Oriented Language with Session Types, pp. 299–318. ISBN: 978-3-540-31483-7. DOI: 10.1007/11580850_16. URL: http://dx.doi.org/10.1007/11580850_16 (cited on p. 162).

Dezani-Ciancaglini, Mariangiola, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou (2006). "Session Types for Object-Oriented Languages". In: *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, pp. 328–352. DOI: 10.1007/11785477_20. URL: http://dx.doi.org/10.1007/11785477_20 (cited on p. 162).

Dezani-Ciancaglini, Mariangiola, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida (2007). "Formal Methods for Components and Objects: 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures". In: Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. Bounded Session Types for Object Oriented Languages, pp. 207–245. ISBN: 978-3-540-74792-5. DOI: 10.1007/978-3-540-74792-5_10. URL: http://dx.doi.org/10.1007/978-3-540-74792-5_10 (cited on p. 162).

Dhaussy, Philippe, Jean-Charles Roger, and Frédéric Boniol (2011). "Reducing State Explosion with Context Modeling for Model-Checking". In: *Proc. of the 13th IEEE International Symposium on High-Assurance Systems Engineering, HASE 2011*, pp. 130–137. DOI: 10.1109/HASE.2011.24. URL: https://doi.org/10.1109/HASE.2011.24 (cited on p. 181).

El Kholy, Warda, Jamal Bentahar, Mohamed El Menshawy, Hongyang Qu, and Rachida Dssouli (2014). "Modeling and verifying choreographed multi-agent-based web service compositions regulated by commitment protocols". In: *Expert systems with applications* 41.16, pp. 7478–7494 (cited on p. 63).

El-Menshawy, Mohamed, Jamal Bentahar, and Rachida Dssouli (2011). "Model Checking Commitment Protocols". In: *Proc. of the 24th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2011*. Ed. by Kishan G. Mehrotra, Chilukuri K. Mohan, Jae C. Oh, Pramod K. Varshney, and Moonis Ali. Vol. 6704. LNCS. Springer, pp. 37–47. DOI: `10.1007/978-3-642-21827-9_5`. URL: `https://doi.org/10.1007/978-3-642-21827-9_5` (cited on p. 63).

Endriss, Ulrich, Nicolas Maudet, Fariba Sadri, and Francesca Toni (2003). "Protocol Conformance for Logic-based Agents". In: *IJCAI*. Morgan Kaufmann, pp. 679–684 (cited on p. 147).

Euzenat, Jérôme and Pavel Shvaiko (2007). *Ontology matching*. Springer. DOI: `10.1007/978-3-540-49612-0`. URL: `https://doi.org/10.1007/978-3-540-49612-0` (cited on p. 145).

Falcone, Yliès (2010). "You Should Better Enforce Than Verify". In: *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*. Ed. by Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann. Vol. 6418. Lecture Notes in Computer Science. Springer, pp. 89–105. DOI: `10.1007/978-3-642-16612-9_9`. URL: `https://doi.org/10.1007/978-3-642-16612-9_9` (cited on p. 53).

Falcone, Yliès, Tom Cornebize, and Jean-Claude Fernandez (2014). "Efficient and Generalized Decentralized Monitoring of Regular Languages". In: *Formal Techniques for Distributed Objects, Components, and Systems: 34th IFIP WG 6.1 International Conference, FORTE 2014. Proceedings*. Ed. by Erika Ábrahám and Catuscia Palamidessi. Springer, pp. 66–83. ISBN: 978-3-662-43613-4. DOI: `10.1007/978-3-662-43613-4_5`. URL: `http://dx.doi.org/10.1007/978-3-662-43613-4_5` (cited on pp. 126, 128).

Ferrando, Angelo (2015). "Parametric protocol-driven agents and their integration in JADE". In: *Proc. of CILC 2015, the 30th Italian Conference on Computational Logic*. Ed. by Davide Ancona, Marco Maratea, and Viviana Mascardi. Vol. 1459. CEUR Workshop Proceedings. CEUR-WS.org, pp. 72–84. URL: `http://ceur-ws.org/Vol-1459/paper26.pdf` (cited on p. 234).

– (2016). "Automatic Partitions Extraction to Distribute the Runtime Verification of a Global Specification". In: *Proceedings of the Doctoral Consortium of AI*IA 2016 co-located with the 15th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2016), Genova, Italy, November 29, 2016*. Ed. by Viviana Mascardi and Ilaria Torre. Vol. 1769. CEUR Workshop Proceedings. CEUR-WS.org, pp. 40–45. URL: `http://ceur-ws.org/Vol-1769/paper07.pdf` (cited on p. 9).

– (2017). "RIVERtools: an IDE for RuntIme VERification of MASs, and Beyond". In: *PRIMA Demo Track 2017* (cited on pp. 10, 195).

– (2019). "The early bird catches the worm: First verify, then monitor!" In: *Science of Computer Programming* 172, pp. 160 –179. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2018.11.008`. URL: `http://`

www.sciencedirect.com/science/article/pii/S0167642318304349 (cited on pp. 10, 159).

Ferrando, Angelo, Davide Ancona, and Viviana Mascardi (2016). "Monitoring Patients with Hypoglycemia Using Self-adaptive Protocol-Driven Agents: A Case Study". In: *Proc. of Engineering Multi-Agent Systems - 4th International Workshop, EMAS 2016, Revised, Selected, and Invited Papers.* Ed. by Matteo Baldoni, Jörg P. Müller, Ingrid Nunes, and Rym Zalila-Wenkstern. Vol. 10093. LNCS. Springer, pp. 39–58. DOI: 10.1007/978-3-319-50983-9_3. URL: http://dx.doi.org/10.1007/978-3-319-50983-9_3 (cited on pp. 11, 126, 219, 221, 224).

– (2017). "Decentralizing MAS Monitoring with DecAMon". In: *Proc. of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017.* Ed. by Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee. ACM, pp. 239–248. URL: http://dl.acm.org/citation.cfm?id=3091164 (cited on pp. 9, 110).

Ferrando, Angelo, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi (2018a). "Recognising Assumption Violations in Autonomous Systems Verification". In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018.* Ed. by Elisabeth André, Sven Koenig, Mehdi Dastani, and Gita Sukthankar. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, pp. 1933–1935. URL: http://dl.acm.org/citation.cfm?id=3238028 (cited on pp. 10, 178).

– (2018b). "Verifying and Validating Autonomous Systems: Towards an Integrated Approach". In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings.* Ed. by Christian Colombo and Martin Leucker. Vol. 11237. Lecture Notes in Computer Science. Springer, pp. 263–281. DOI: 10.1007/978-3-030-03769-7_15. URL: https://doi.org/10.1007/978-3-030-03769-7_15 (cited on p. 178).

Finin, Tim, Richard Fritzson, Don McKay, and Robin McEntire (1994). "KQML As an Agent Communication Language". In: *Proc. of the Third International Conference on Information and Knowledge Management.* CIKM '94. Gaithersburg, Maryland, USA: ACM, pp. 456–463. ISBN: 0-89791-674-3. DOI: 10.1145/191246.191322. URL: http://doi.acm.org/10.1145/191246.191322 (cited on p. 18).

Fisher, Michael and Chiara Ghidini (2010). "Executable specifications of resource-bounded agents". In: *Autonomous Agents and Multi-Agent Systems* 21.3, pp. 368–396. DOI: 10.1007/s10458-009-9105-x. URL: https://doi.org/10.1007/s10458-009-9105-x (cited on p. 21).

Fisher, Michael and Anthony Hepple (2009). "Executing Logical Agent Specifications". In: *Multi-Agent Programming, Languages, Tools and Applications.* Ed. by Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni. Springer, pp. 1–27. DOI: 10.1007/978-0-387-89299-3_1. URL: https://doi.org/10.1007/978-0-387-89299-3_1 (cited on p. 21).

Fisher, Michael and Michael Wooldridge (1997). “On the Formal Specification and Verification of Multi-Agent Systems”. In: *Int. J. Cooperative Inf. Syst.* 6.1, pp. 37–66 (cited on p. 3).

Fornara, Nicoletta and Marco Colombetti (2003). “Protocol specification using a commitment based ACL”. In: *Proc. of the 2003 Workshop on Agent Communication Languages.* Vol. 2922. LNCS. Springer, pp. 108–127 (cited on p. 63).

Foundation for Intelligent Physical Agents (2001). “FIPA English Auction Interaction Protocol Specification”. `http://www.fipa.org/specs/fipa00031/XC00031F.pdf` (cited on p. 74).

Fraigniaud, Pierre, Sergio Rajsbaum, and Corentin Travers (2014). “On the Number of Opinions Needed for Fault-Tolerant Run-Time Monitoring in Distributed Systems”. In: *Runtime Verification: 5th International Conference, RV 2014. Proceedings.* Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Springer, pp. 92–107. ISBN: 978-3-319-11164-3. DOI: `10.1007/978-3-319-11164-3_9`. URL: `http://dx.doi.org/10.1007/978-3-319-11164-3_9` (cited on p. 128).

Fraigniaud, Pierre, Sergio Rajsbaum, Matthieu Roy, and Corentin Travers (2014). “The Opinion Number of Set-Agreement”. In: *Principles of Distributed Systems: 18th International Conference, OPODIS 2014. Proceedings.* Ed. by Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro. Springer, pp. 155–170. ISBN: 978-3-319-14472-6. DOI: `10.1007/978-3-319-14472-6_11`. URL: `http://dx.doi.org/10.1007/978-3-319-14472-6_11` (cited on p. 128).

García-Ojeda, Juan C., Scott A. DeLoach, and Robby (2009). “AgentTool III: from process definition to code generation”. In: *Proc. of AAMAS 2009.* IFAAMAS, pp. 1393–1394. DOI: `10.1145/1558109.1558311`. URL: `http://doi.acm.org/10.1145/1558109.1558311` (cited on p. 98).

Gay, Simon J., Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira (Jan. 2010). “Modular Session Types for Distributed Object-oriented Programming”. In: *SIGPLAN Not.* 45.1, pp. 299–312. ISSN: 0362-1340. DOI: `10.1145/1707801.1706335`. URL: `http://doi.acm.org/10.1145/1707801.1706335` (cited on p. 162).

Gensini, Gian Franco, Camilla Alderighi, Raffaele Rasoini, Marco Mazzanti, and Giancarlo Casolo (2017). “Value of Telemonitoring and Telemedicine in Heart Failure Management”. In: *Cardiac Failure Review* 3.2 (cited on p. 220).

Gerasimou, Simos, Radu Calinescu, Stepan Shevtsov, and Danny Weyns (2017). “UNDERSEA: An Exemplar for Engineering Self-Adaptive Unmanned Underwater Vehicles”. In: *Proc. of the 12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2017.* IEEE Computer Society, pp. 83–89. DOI: `10.1109/SEAMS.2017.19`. URL: `https://doi.org/10.1109/SEAMS.2017.19` (cited on p. 66).

Giordano, Laura and Alberto Martelli (2007). “Verifying Agent Conformance with Protocols Specified in a Temporal Action Logic”. In: *AI*IA.* Vol. 4733. LNCS, pp. 145–156 (cited on p. 147).

Gluch, David, Santiago Comella-Dorda, John Hudak, Grace Lewis, and Chuck Weinstock (Feb. 2019). “Model-Based Verification: Guidelines for Generating Expected Properties”. In: (cited on p. 28).

Gottlob, Georg (2012). “On minimal constraint networks”. In: *Artif. Intell.* 191-192, pp. 42–60. DOI: 10.1016/j.artint.2012.07.006. URL: http://dx.doi.org/10.1016/j.artint.2012.07.006 (cited on p. 126).

Gruber, Thomas R. (June 1993). “A Translation Approach to Portable Ontology Specifications”. In: *Knowl. Acquis.* 5.2, pp. 199–220. ISSN: 1042-8143. DOI: 10.1006/knac.1993.1008. URL: http://dx.doi.org/10.1006/knac.1993.1008 (cited on p. 19).

Gruber, Tom (2009). “Ontology”. In: *Encyclopedia of Database Systems.* Ed. by Ling Liu and M. Tamer Özsu. Springer US, pp. 1963–1965. DOI: 10.1007/978-0-387-39940-9_1318. URL: https://doi.org/10.1007/978-0-387-39940-9_1318 (cited on p. 19).

Gui, Lin, Jun Sun, Yang Liu, Yuanjie Si, Jin Song Dong, and Xinyu Wang (2013). “Combining model checking and testing with an application to reliability prediction and distribution”. In: *International Symposium on Software Testing and Analysis, ISSTA ’13, Lugano, Switzerland, July 15-20, 2013*, pp. 101–111. DOI: 10.1145/2483760.2483779. URL: http://doi.acm.org/10.1145/2483760.2483779 (cited on pp. 10, 162).

Günay, Akin, Yang Liu, and Jie Zhang (Sept. 2016). “PROMOCA: Probabilistic Modeling and Analysis of Agents in Commitment Protocols”. In: *J. Artif. Int. Res.* 57.1, pp. 465–508. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=3176748.3176759 (cited on p. 63).

Havelund, K., G. Reger, and G. Rosu (2018). “Runtime Verification – Past Experiences and Future Projections”. In: *Special issue in celebration of issue number 10,000 of Lecture Notes in Computer Science.* Vol. 10000. Lecture Notes in Computer Science (cited on p. 80).

Havelund, Klaus (1999). “Java PathFinder, A Translator from Java to Promela”. In: *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings.* Ed. by Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink. Vol. 1680. Lecture Notes in Computer Science. Springer, p. 152. DOI: 10.1007/3-540-48234-2_11. URL: https://doi.org/10.1007/3-540-48234-2_11 (cited on p. 35).

Havelund, Klaus, Michael R. Lowry, and John Penix (2001). “Formal Analysis of a Space-Craft Controller Using SPIN”. In: *IEEE Trans. Software Eng.* 27.8, pp. 749–765 (cited on p. 3).

Havelund, Klaus and Thomas Pressburger (2000). “Model Checking JAVA Programs using JAVA PathFinder”. In: *STTT* 2.4, pp. 366–381. DOI: 10.1007/s100090050043. URL: https://doi.org/10.1007/s100090050043 (cited on p. 35).

Havelund, Klaus, Martin Leucker, Martin Sachenbacher, Oleg Sokolsky, and Brian C. Williams, eds. (2010). *Runtime Verification, Diagnosis, Planning and Control for Autonomous Systems, 07.11. - 12.11.2010.* Vol. 10451. Dagstuhl Sem-

inar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (cited on p. 80).

Hendler, James (Mar. 2001). "Agents and the Semantic Web". In: *IEEE Intelligent Systems* 16.2, pp. 30–37. ISSN: 1541-1672. DOI: `10.1109/5254.920597`. URL: `http://dx.doi.org/10.1109/5254.920597` (cited on p. 19).

Henzinger, Thomas A. (1996). "The Theory of Hybrid Automata". In: *Proc. of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pp. 278–292 (cited on p. 181).

Herlihy, Maurice (Jan. 1991). "Wait-free Synchronization". In: *ACM Trans. Program. Lang. Syst.* 13.1, pp. 124–149. ISSN: 0164-0925. DOI: `10.1145/114005.102808`. URL: `http://doi.acm.org/10.1145/114005.102808` (cited on p. 128).

Higuera, Colin de la (2010). *Grammatical Inference: Learning Automata and Grammars.* New York, NY, USA: Cambridge University Press. ISBN: 0521763169, 9780521763165 (cited on p. 244).

Hindriks, Koen V., Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer (1997). "Formal Semantics for an Abstract Agent Programming Language". In: *Intelligent Agents IV, Agent Theories, Architectures, and Languages, 4th International Workshop, ATAL '97, Providence, Rhode Island, USA, July 24-26, 1997, Proceedings.* Ed. by Munindar P. Singh, Anand S. Rao, and Michael Wooldridge. Vol. 1365. Lecture Notes in Computer Science. Springer, pp. 215–229. DOI: `10.1007/BFb0026761`. URL: `https://doi.org/10.1007/BFb0026761` (cited on p. 21).

– (2000). "Agent Programming with Declarative Goals". In: *Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop, ATAL 2000, Boston, MA, USA, July 7-9, 2000, Proceedings.* Ed. by Cristiano Castelfranchi and Yves Lespérance. Vol. 1986. Lecture Notes in Computer Science. Springer, pp. 228–243. DOI: `10.1007/3-540-44631-1_16`. URL: `https://doi.org/10.1007/3-540-44631-1_16` (cited on p. 21).

Hinrichs, Timothy L., A. Prasad Sistla, and Lenore D. Zuck (2014). "Model Check What You Can, Runtime Verify the Rest". In: *HOWARD-60.* Vol. 42. EPiC Series in Computing. EasyChair, pp. 234–244 (cited on p. 160).

Holzmann, Gerard J. (1991). *Design and Validation of Computer Protocols.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0-13-539925-4 (cited on pp. 161, 174).

– (1997). "The Model Checker SPIN". In: *IEEE Trans. Software Eng.* 23.5, pp. 279–295. DOI: `10.1109/32.588521`. URL: `https://doi.org/10.1109/32.588521` (cited on pp. 161, 174).

– (2002). "Software Analysis and Model Checking". In: *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings.* Ed. by Ed Brinksma and Kim Guldstrand Larsen. Vol. 2404. Lecture Notes in Computer Science. Springer, pp. 1–16. DOI: `10.1007/3-540-45657-0_1`. URL: `https://doi.org/10.1007/3-540-45657-0_1` (cited on p. 160).

Holzmann, Gerard J. (2004). *The SPIN Model Checker - primer and reference manual.* Addison-Wesley. ISBN: 978-0-321-22862-8 (cited on pp. 161, 174).

Honda, Kohei, Nobuko Yoshida, and Marco Carbone (2008). "Multiparty asynchronous session types". In: *Proc. of POPL 2008.* ACM, pp. 273–284 (cited on p. 98).

Hopcroft, John E. and Jeffrey D. Ullman (1969). *Formal languages and their relation to automata.* Addison-Wesley series in computer science and information processing. Addison-Wesley (cited on p. 165).

Hu, Raymond, Nobuko Yoshida, and Kohei Honda (2008). "ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings". In: Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. Session-Based Distributed Programming in Java, pp. 516–541. ISBN: 978-3-540-70592-5. DOI: 10.1007/978-3-540-70592-5_22. URL: http://dx.doi.org/10.1007/978-3-540-70592-5_22 (cited on p. 162).

Huget, Marc-Philippe and James Odell (2004). "Representing Agent Interaction Protocols with Agent UML". In: *Proc. of AAMAS 2004.* IEEE Computer Society, pp. 1244–1245 (cited on p. 98).

"IEEE Standard for Software Verification and Validation" (2005). In: *IEEE Std 1012-2004 (Revision of IEEE Std 1012-1998)*, pp. 1–110. DOI: 10.1109/IEEESTD.2005.96278 (cited on p. 30).

Iftikhar, M. Usman and Danny Weyns (2012). "A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System". In: *Proc. of the 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012.* Ed. by Natallia Kokash and António Ravara. Vol. 91. EPTCS, pp. 45–62. DOI: 10.4204/EPTCS.91.4. URL: https://doi.org/10.4204/EPTCS.91.4 (cited on pp. 65, 66).

Iftikhar, M Usman and Danny Weyns (2016). "Towards runtime statistical model checking for self-adaptive systems". Tech. Report CW 693, August 2016, Department of Computer Science, KU Leuven (cited on pp. 65, 66).

Iftikhar, M. Usman and Danny Weyns (2017). "ActivFORMS: A Runtime Environment for Architecture-Based Adaptation with Guarantees". In: *Proc. of the 2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017.* IEEE, pp. 278–281. DOI: 10.1109/ICSAW.2017.21. URL: https://doi.org/10.1109/ICSAW.2017.21 (cited on p. 65).

Iftikhar, Muhammad Usman, Gowri Sankar Ramachandran, Pablo Bollansée, Danny Weyns, and Danny Hughes (2017). "DeltaIoT: A Self-Adaptive Internet of Things Exemplar". In: *Proc. of the 12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2017.* IEEE Computer Society, pp. 76–82. DOI: 10.1109/SEAMS.2017.21. URL: https://doi.org/10.1109/SEAMS.2017.21 (cited on p. 66).

Iglesia, Didac Gil de la and Danny Weyns (2015). "MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems". In: *ACM Trans. Auton. Adapt. Syst.* 10.3, 15:1–15:31. DOI: 10.1145/2724719. URL: http://doi.acm.org/10.1145/2724719 (cited on p. 65).

Jennings, Nicholas R., Katia P. Sycara, and Michael Wooldridge (1998). "A Roadmap of Agent Research and Development". In: *Autonomous Agents and Multi-Agent Systems* 1.1, pp. 7–38. DOI: 10.1023/A:1010090405266. URL: http://dx.doi.org/10.1023/A:1010090405266 (cited on p. 16).

Jezewski, Janusz, Adam Pawlak, Krzysztof Horoba, Janusz Wrobel, Robert Czabanski, and Michal Jezewski (2016). "Selected design issues of the medical cyber-physical system for telemonitoring pregnancy at home". In: *Microprocessors and Microsystems* 46.Part A, pp. 35 –43. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2016.07.005. URL: http://www.sciencedirect.com/science/article/pii/S0141933116300874 (cited on p. 220).

Jiao, Wenpin and Yanchun Sun (2016). "Self-adaptation of multi-agent systems in dynamic environments based on experience exchanges". In: *Journal of Systems and Software* 122.Supplement C, pp. 165 –179. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.09.025. URL: http://www.sciencedirect.com/science/article/pii/S0164121216301844 (cited on p. 59).

Joshi, Yogi, Guy Martin Tchamgoue, and Sebastian Fischmeister (2017). "Runtime verification of LTL on lossy traces". In: *Proc. of SAC 2017*. ACM, pp. 1379–1386 (cited on pp. 80, 101).

Kacprzak, Magdalena, Alessio Lomuscio, and Wojciech Penczek (2004). "Verification of Multiagent Systems via Unbounded Model Checking". In: *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), 19-23 August 2004, New York, NY, USA*. IEEE Computer Society, pp. 638–645. DOI: 10.1109/AAMAS.2004.10086. URL: http://doi.ieeecomputersociety.org/10.1109/AAMAS.2004.10086 (cited on p. 35).

Kamali, Maryam, Louise A. Dennis, Owen McAree, Michael Fisher, and Sandor M. Veres (2017). "Formal verification of autonomous vehicle platooning". In: *Science of Computer Programming*, pp. –. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2017.05.006. URL: http://www.sciencedirect.com/science/article/pii/S0167642317301168 (cited on p. 184).

Kavantzas, Nickolas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto (2005). "Web Services Choreography Description Language v. 1.0" (cited on p. 146).

King, Thomas C., Akin Günay, Amit K. Chopra, and Munindar P. Singh (2017). "Tosca: Operationalizing Commitments Over Information Protocols". In: *Proc. of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017*. Ed. by Carles Sierra. ijcai.org, pp. 256–264. DOI: 10.24963/ijcai.2017/37. URL: https://doi.org/10.24963/ijcai.2017/37 (cited on pp. 61, 63).

Kraai, Imke, Arjen de Vries, Karin Vermeulen, Vincent van Deursen, Martje van der Wal, Richard de Jong, René van Dijk, Tiny Jaarsma, Hans Hillege, and Ivonne Lesman (2016). "The value of telemonitoring and ICT-guided disease

management in heart failure: Results from the IN TOUCH study". In: *International Journal of Medical Informatics* 85.1, pp. 53 –60. ISSN: 1386-5056. DOI: https://doi.org/10.1016/j.ijmedinf.2015.10.001. URL: http://www.sciencedirect.com/science/article/pii/S1386505615300447 (cited on p. 220).

Kucher, Kostiantyn and Danny Weyns (2013). "A Self-Adaptive Software System to Support Elderly Care". In: *Proc. of MIT 2013, Modern Information Technology* (cited on p. 221).

Ladkin, Peter B and Stefan Leue (1995). "Interpreting message flow graphs". In: *Formal Aspects of Computing* 7.5, pp. 473–509 (cited on p. 102).

Lamport, Leslie (July 1978). "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7, pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563 (cited on p. 130).

Lanese, Ivan, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro (2008). "Bridging the gap between interaction-and process-oriented choreographies". In: *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*. IEEE, pp. 323–332 (cited on pp. 98, 102, 103).

Leotta, Maurizio, Diego Clerissi, Dario Olianas, Filippo Ricca, Davide Ancona, Giorgio Delzanno, Luca Franceschini, and Marina Ribaudo (2018). "An acceptance testing approach for Internet of Things systems". In: *IET Software* 12.5, pp. 430–436 (cited on pp. 236, 238).

Leucker, Martin and Christian Schallhart (2009). "A brief account of runtime verification". In: *The Journal of Logic and Algebraic Programming* 78.5. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07), pp. 293 –303. ISSN: 1567-8326. DOI: http://dx.doi.org/10.1016/j.jlap.2008.08.004. URL: http://www.sciencedirect.com/science/article/pii/S1567832608000775 (cited on pp. 3, 29, 30, 80, 163).

Lomuscio, Alessio and Franco Raimondi (2006). "MCMAS: A Model Checker for Multi-agent Systems". In: *Proc. of Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006*, pp. 450–454. DOI: 10.1007/11691372_31. URL: https://doi.org/10.1007/11691372_31 (cited on p. 182).

Luo, Q., Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Serbanuta, and G. Rosu (2014a). "RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties". In: *RV'14*. Vol. 8734. Springer, pp. 285–300 (cited on p. 31).

Luo, Qingzhou, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian-Florin Serbanuta, and Grigore Rosu (2014b). "RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties". In: *Runtime Verification, RV 2014*, pp. 285–300 (cited on p. 69).

Macías-Escrivá, Frank D., Rodolfo E. Haber, Raúl M. del Toro, and Vicente Hernández (2013). "Self-adaptive systems: A survey of current approaches,

research challenges and applications". In: *Expert Syst. Appl.* 40.18, pp. 7267–7279 (cited on p. 2).

Maler, Oded and Dejan Nickovic (2004). "Monitoring Temporal Properties of Continuous Signals." In: *FORMATS/FTRTFT*. Ed. by Yassine Lakhnech and Sergio Yovine. Vol. 3253. Lecture Notes in Computer Science. Springer, pp. 152–166. URL: `http://dblp.uni-trier.de/db/conf/formats/formats2004.html#MalerN04` (cited on p. 181).

Mascardi, Viviana and Davide Ancona (2013). "Attribute Global Types for Dynamic Checking of Protocols in Logic-based Multiagent Systems". In: *Theory and Practice of Logic Programming* 13.4-5-Online-Supplement (cited on pp. 37, 234).

Mascardi, Viviana, Daniela Briola, and Davide Ancona (2013). "On the Expressiveness of Attribute Global Types: The Formalization of a Real Multiagent System Protocol". In: *Proc. of AI*IA 2013: Advances in Artificial Intelligence - XIIIth International Conference of the Italian Association for Artificial Intelligence*, pp. 300–311. DOI: `10.1007/978-3-319-03524-6_26`. URL: `http://dx.doi.org/10.1007/978-3-319-03524-6_26` (cited on pp. 234, 235).

Mascardi, Viviana, Angela Locoro, and Paolo Rosso (2010). "Automatic Ontology Matching via Upper Ontologies: A Systematic Evaluation". In: *IEEE Trans. Knowl. Data Eng.* 22.5, pp. 609–623 (cited on p. 240).

Mascardi, Viviana, Davide Ancona, Rafael H. Bordini, and Alessandro Ricci (2011). "CooL-AgentSpeak: Enhancing AgentSpeak-DL Agents with Plan Exchange and Ontology Services". In: *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 02*. WI-IAT '11. Washington, DC, USA: IEEE Computer Society, pp. 109–116. ISBN: 978-0-7695-4513-4. DOI: `10.1109/WI-IAT.2011.255`. URL: `https://doi.org/10.1109/WI-IAT.2011.255` (cited on p. 19).

Meron, Denis and Bruno Mermet (2006). "A Tool Architecture to Verify Properties of Multiagent System at Runtime". In: *Proc. of PROMAS*. Vol. 4411. LNCS. Springer, pp. 201–216 (cited on p. 196).

Merwe, Heila van der, Brink van der Merwe, and Willem Visser (2012). "Verifying android applications using Java PathFinder". In: *ACM SIGSOFT Software Engineering Notes* 37.6, pp. 1–5. DOI: `10.1145/2382756.2382797`. URL: `http://doi.acm.org/10.1145/2382756.2382797` (cited on p. 180).

Merz, Stephan (2000). "Model Checking: A Tutorial Overview". In: *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000*. Ed. by Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan. Vol. 2067. Lecture Notes in Computer Science. Springer, pp. 3–38. DOI: `10.1007/3-540-45510-8_1`. URL: `https://doi.org/10.1007/3-540-45510-8_1` (cited on p. 160).

– (2001). "Model checking: A tutorial overview". In: *Modeling and Verification of Parallel Processes*. Springer-Verlag, pp. 3–38 (cited on pp. 3, 28).

Milner, Robin and Mads Tofte (1991). "Co-induction in relational semantics". In: *Theoretical Computer Science* 87.1, pp. 209 –220. ISSN: 0304-3975. DOI:

https://doi.org/10.1016/0304-3975(91)90033-X. URL: http://www.sciencedirect.com/science/article/pii/030439759190033X (cited on p. 54).

Montanari, Ugo (1974). "Networks of constraints: Fundamental properties and applications to picture processing". In: *Information Sciences* 7, pp. 95–132 (cited on p. 126).

Moreira, Álvaro F., Renata Vieira, Rafael H. Bordini, and Jomi F. Hübner (2006). "Agent-Oriented Programming with Underlying Ontological Reasoning". In: *Declarative Agent Languages and Technologies III*. Ed. by Matteo Baldoni, Ulle Endriss, Andrea Omicini, and Paolo Torroni. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 155–170. ISBN: 978-3-540-33107-0 (cited on p. 19).

Mostafa, M. and B. Bonakdarpour (2015). "Decentralized Runtime Verification of LTL Specifications in Distributed Systems". In: *Parallel and Distributed Processing Symposium, IEEE International Conference, IPDPS 2015. Proceedings*, pp. 494–503. DOI: 10.1109/IPDPS.2015.95 (cited on pp. 125, 128).

Muscettola, Nicola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams (1998). "Remote Agent: To Boldly Go Where No AI System Has Gone Before". In: *Artif. Intell.* 103.1-2, pp. 5–47 (cited on p. 3).

Myers, Glenford J. and Corey Sandler (2004). *The Art of Software Testing*. John Wiley & Sons. ISBN: 0471469122 (cited on p. 30).

Myers, Glenford J., Corey Sandler, and Tom Badgett (2011). *The Art of Software Testing*. 3rd. Wiley Publishing. ISBN: 1118031962, 9781118031964 (cited on p. 160).

Nguyen, Luan Viet, Christian Schilling, Sergiy Bogomolov, and Taylor T. Johnson (2015). "Runtime Verification for Hybrid Analysis Tools". In: *Proc. of Runtime Verification: 6th International Conference, RV 2015*, pp. 281–286. ISBN: 978-3-319-23820-3. DOI: 10.1007/978-3-319-23820-3_19. URL: http://dx.doi.org/10.1007/978-3-319-23820-3_19 (cited on p. 181).

Omicini, Andrea, Alessandro Ricci, and Mirko Viroli (2008). "Artifacts in the A&A meta-model for multi-agent systems". In: *Autonomous Agents and Multi-Agent Systems* 17.3, pp. 432–456 (cited on p. 146).

Padgham, Lin and Michael Winikoff (2002). "Prometheus: A Methodology for Developing Intelligent Agents". In: *Proc. of AAMAS 2002*. Bologna, Italy: ACM, pp. 37–38. ISBN: 1-58113-480-0. DOI: 10.1145/544741.544749. URL: http://doi.acm.org/10.1145/544741.544749 (cited on p. 98).

Papazoglou, Mike P. (2003). "Service -Oriented Computing: Concepts, Characteristics and Directions". In: *Proc. of WISE 2003*. IEEE Computer Society, pp. 3–. ISBN: 0-7695-1999-7. URL: http://dl.acm.org/citation.cfm?id=960322.960404 (cited on p. 98).

Parekh, Rajesh and Vasant Honavar (1998). "Grammar inference, automata induction, and language acquisition". In: *Handbook of Natural Language Processing*. Marcel Dekker, pp. 727–764 (cited on p. 244).

Pechoucek, Michal and Vladimír Marík (2008). "Industrial deployment of multi-agent technologies: review and selected case studies". In: *Autonomous*

Agents and Multi-Agent Systems 17.3, pp. 397–431. DOI: 10.1007/s10458-008-9050-0. URL: https://doi.org/10.1007/s10458-008-9050-0 (cited on p. 2).

Penix, John, Willem Visser, Eric Engstrom, Aaron Larson, and Nicholas Weininger (2000). “Verification of time partitioning in the DEOS scheduler kernel”. In: *Proc. of the 22nd International Conference on Software Engineering*, pp. 488–497 (cited on p. 181).

Pnueli, Amir (1977). “The temporal logic of programs”. In: *Proc. of the 18th Annual Symposium on Foundations of Computer Science*, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cited on pp. 31, 53, 82, 160, 235).

Pokahr, Alexander, Lars Braubach, and Winfried Lamersdorf (2005). “Jadex: A BDI Reasoning Engine”. In: *Multi-Agent Programming: Languages, Platforms and Applications.* Ed. by Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni. Vol. 15. Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, pp. 149–174 (cited on p. 21).

Poslad, Stefan (2009). *Ubiquitous Computing: Smart Devices, Environments and Interactions.* 1st. Wiley Publishing. ISBN: 0470035609, 9780470035603 (cited on p. 242).

Rabiner, Lawrence R. (1990). “Readings in Speech Recognition”. In: ed. by Alex Waibel and Kai-Fu Lee. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Chap. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, pp. 267–296. ISBN: 1-55860-124-4. URL: http://dl.acm.org/citation.cfm?id=108235.108253 (cited on pp. 36, 82).

Rabiner, Lawrence R. and Biing-Hwang Juang (1986). “An introduction to hidden Markov models”. In: *IEEE ASSP Magazine*, pp. 4–16 (cited on p. 36).

Raimondi, Franco and Alessio Lomuscio (2007). “Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams”. In: *J. Applied Logic* 5.2, pp. 235–251. DOI: 10.1016/j.jal.2005.12.010. URL: https://doi.org/10.1016/j.jal.2005.12.010 (cited on pp. 35, 182).

Rao, Anand S. (1996). “AgentSpeak(L): BDI Agents Speak out in a Logical Computable Language”. In: *Proc. of the 7th European Workshop on Modelling Autonomous Agents in a Multi-agent World : Agents Breaking Away: Agents Breaking Away.* MAAMAW ’96. Einhoven, The Netherlands: Springer-Verlag New York, Inc., pp. 42–55. ISBN: 3-540-60852-4. URL: http://dl.acm.org/citation.cfm?id=237945.237953 (cited on p. 20).

Rao, Anand S. and Michael P. Georgeff (1992). “An Abstract Architecture for Rational Agents”. In: *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR’92). Cambridge, MA, USA, October 25-29, 1992.* Ed. by Bernhard Nebel, Charles Rich, and William R. Swartout. Morgan Kaufmann, pp. 439–449 (cited on p. 20).

Ricci, Alessandro, Michele Piunti, and Mirko Viroli (2011). “Environment programming in multi-agent systems: an artifact-based perspective”. In:

Autonomous Agents and Multi-Agent Systems 23.2, pp. 158–192 (cited on p. 146).

Riemsdijk, Birna van, Wiebe van der Hoek, and John-Jules Ch. Meyer (2003). "Agent programming in dribble: from beliefs to goals using plans". In: *The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14-18, 2003, Melbourne, Victoria, Australia, Proceedings*. ACM, pp. 393–400. DOI: 10.1145/860575.860639. URL: http://doi.acm.org/10.1145/860575.860639 (cited on p. 21).

Sangiorgi, Davide (May 2009). "On the Origins of Bisimulation and Coinduction". In: *ACM Trans. Program. Lang. Syst.* 31.4, 15:1–15:41. ISSN: 0164-0925. DOI: 10.1145/1516507.1516510. URL: http://doi.acm.org/10.1145/1516507.1516510 (cited on p. 54).

Searle, John R (1969). *Speech acts: An essay in the philosophy of language*. Vol. 626. Cambridge university press (cited on p. 18).

Singh, Munindar P. (2011a). "Information-driven interaction-oriented programming: BSPL, the blindingly simple protocol language". In: *Proc. of the 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011)*. Ed. by Liz Sonenberg, Peter Stone, Kagan Tumer, and Pinar Yolum. IFAAMAS, pp. 491–498. URL: http://portal.acm.org/citation.cfm?id=2031687&CFID=54178199&CFTOKEN=61392764 (cited on pp. 61, 102, 129).

Singh, Munindar P. (2011b). "LoST: Local State Transfer - An Architectural Style for the Distributed Enactment of Business Protocols". In: *Proc. of the IEEE International Conference on Web Services, ICWS 2011*. IEEE Computer Society, pp. 57–64. DOI: 10.1109/ICWS.2011.48. URL: https://doi.org/10.1109/ICWS.2011.48 (cited on p. 62).

– (2012). "Semantics and verification of information-based protocols". In: *Proc. of the International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012*. Ed. by Wiebe van der Hoek, Lin Padgham, Vincent Conitzer, and Michael Winikoff. IFAAMAS, pp. 1149–1156. URL: http://dl.acm.org/citation.cfm?id=2343861 (cited on p. 62).

Singh, Munindar P (2014). "Bliss: Specifying declarative service protocols". In: *Proc. of Services Computing (SCC), 2014 IEEE International Conference*. IEEE Computer Society, pp. 235–242 (cited on p. 62).

Sistla, A. Prasad, Miloš Žefran, and Yao Feng (2012). "Runtime Monitoring of Stochastic Cyber-physical Systems with Hybrid State". In: *Proc. of the Second International Conference on Runtime Verification*. RV'11, pp. 276–293 (cited on p. 181).

Staiger, Ludwig (1997). "On omega-power Languages". In: *New Trends in Formal Languages*. Vol. 1218. Lecture Notes in Computer Science. Springer, pp. 377–394 (cited on p. 165).

Steegmans, Elke, Danny Weyns, Tom Holvoet, and Yolande Berbers (2004). "A Design Process for Adaptive Behavior of Situated Agents". In: *Proc. of Agent-Oriented Software Engineering V, Revised Selected Papers*. Ed. by James Odell,

Paolo Giorgini, and Jörg P. Müller. Vol. 3382. LNCS. Springer, pp. 109–125 (cited on p. 65).

Steventon, Adam et al. (2012). “Effect of telehealth on use of secondary care and mortality: findings from the Whole System Demonstrator cluster randomised trial”. In: *British Medical Journal* 344. DOI: 10.1136/bmj.e3874. eprint: http://www.bmj.com/content/344/bmj.e3874.full.pdf (cited on p. 220).

Stoller, Scott D., Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok (2011). “Runtime Verification with State Estimation”. In: *Proc. of RV 2011, Revised Selected Papers.* Vol. 7186. LNCS. Springer, pp. 193–207 (cited on pp. 80–82, 85, 90, 100, 129).

Testerink, Bas, Nils Bulling, and Mehdi Dastani (2016). “Security and Robustness for Collaborative Monitors”. In: *Coordination, Organizations, Institutions, and Normes in Agent Systems XI - COIN 2015 International Workshops, Revised Selected Papers.* Ed. by Virginia Dignum, Pablo Noriega, Murat Sensoy, and Jaime Simão Sichman. Vol. 9628. Lecture Notes in Computer Science. Springer, pp. 376–395. ISBN: 978-3-319-42690-7. DOI: 10.1007/978-3-319-42691-4. URL: http://dx.doi.org/10.1007/978-3-319-42691-4 (cited on p. 125).

Testerink, Bas, Mehdi Dastani, and Nils Bulling (2016). “Distributed Controllers for Norm Enforcement”. In: *ECAI 2016 - 22nd European Conference on Artificial Intelligence – Including Prestigious Applications of Artificial Intelligence (PAIS 2016).* Ed. by Gal A. Kaminka, Maria Fox, Paolo Bouquet, Eyke Hüllermeier, Virginia Dignum, Frank Dignum, and Frank van Harmelen. Vol. 285. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 751–759 (cited on p. 125).

The OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee (2007). “Web Services Business Process Execution Language Version 2.0” (cited on p. 146).

Thomas, Wolfgang (1990). “Automata on Infinite Objects”. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pp. 133–192 (cited on p. 165).

Tinnemeier, Nick A. M., Mehdi Dastani, John-Jules Ch. Meyer, and Leendert W. N. van der Torre (2009). “Programming Normative Artifacts with Declarative Obligations and Prohibitions”. In: *Proc. of IAT 2009.* IEEE Computer Society, pp. 145–152 (cited on p. 100).

Tkachuk, Oksana, Matthew B. Dwyer, and Corina S. Pasareanu (2003). “Automated Environment Generation for Software Model Checking”. In: *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pp. 116–129. DOI: 10.1109/ASE.2003.1240300. URL: https://doi.org/10.1109/ASE.2003.1240300 (cited on p. 181).

Torroni, Paolo, Pinar Yolum, Munindar P. Singh, Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, and Paola Mello (2009). “Modelling Interactions via Commitments and Expectations”. In: *Handbook of Research*

on Multi-Agent Systems: Semantics and Dynamics of Organizational Models (cited on p. 182).

UK Department of Health (2011). "Whole System Demonstrator Programme – Headline Findings" (cited on p. 220).

Vardi, Moshe Y. (2007). "Automata-Theoretic Model Checking Revisited". In: *VMCAI.* Vol. 4349. Lecture Notes in Computer Science. Springer, pp. 137–150 (cited on p. 161).

Venkatraman, Mahadevan and Munindar P. Singh (1999). "Verifying Compliance with Commitment Protocols". In: *Autonomous Agents and Multi-Agent Systems* 2.3, pp. 217–236. DOI: 10.1023/A:1010056221226. URL: https://doi.org/10.1023/A:1010056221226 (cited on p. 63).

Visser, Willem, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda (2003). "Model Checking Programs". In: *Autom. Softw. Eng.* 10.2, pp. 203–232. DOI: 10.1023/A:1022920129859. URL: https://doi.org/10.1023/A:1022920129859 (cited on pp. 35, 160).

Wallace, Eric L. et al. (2017). "Remote Patient Management for Home Dialysis Patients". In: *Kidney International Reports.* ISSN: 2468-0249. DOI: https://doi.org/10.1016/j.ekir.2017.07.010. URL: http://www.sciencedirect.com/science/article/pii/S2468024917303170 (cited on p. 220).

Weiss, Gerhard, ed. (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* Cambridge, MA, USA: MIT Press. ISBN: 0-262-23203-0 (cited on p. 16).

Weyns, Danny (2012). "Towards an integrated approach for validating qualities of self-adaptive systems". In: *Proc. of the International Workshop on Dynamic Analysis.* Ed. by Eric Bodden and Madanlal Musuvathi. ACM, pp. 24–29. DOI: 10.1145/2338966.2336803. URL: http://doi.acm.org/10.1145/2338966.2336803 (cited on p. 65).

– (2018). "Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges". In: *Handbook of Software Engineering.* Ed. by Richard Taylor, Kyo Chul Kang, and Sungdeok Cha. Springer (cited on pp. 59, 60, 232).

Weyns, Danny and Radu Calinescu (2015). "Tele Assistance: A Self-Adaptive Service-Based System Exemplar". In: *Proc. of the 10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015.* Ed. by Paola Inverardi and Bradley R. Schmerl. IEEE Computer Society, pp. 88–92. DOI: 10.1109/SEAMS.2015.27. URL: https://doi.org/10.1109/SEAMS.2015.27 (cited on p. 66).

Weyns, Danny and Michael Georgeff (2010). "Self-Adaptation Using Multiagent Systems". In: *IEEE Software* 27.1, pp. 86–91 (cited on p. 59).

Weyns, Danny and M. Usman Iftikhar (2016). "Model-Based Simulation at Runtime for Self-Adaptive Systems". In: *Proc. of the 2016 IEEE International Conference on Autonomic Computing, ICAC.* Ed. by Samuel Kounev, Holger Giese, and Jie Liu. IEEE Computer Society, pp. 364–373. DOI: 10.1109/

ICAC.2016.67. URL: https://doi.org/10.1109/ICAC.2016.67 (cited on p. 65).

Weyns, Danny, Kurt Schelfthout, and Tom Holvoet (2005). "Architectural design of a distributed application with autonomic quality requirements". In: *ACM SIGSOFT Software Engineering Notes* 30.4, pp. 1–7. DOI: 10.1145/1082983.1083076. URL: http://doi.acm.org/10.1145/1082983.1083076 (cited on p. 65).

Weyns, Danny, M. Usman Iftikhar, Danny Hughes, and Nelson Matthys (2018). "Applying Architecture-Based Adaptation to Automate the Management of Internet-of-Things". In: *ECSA*. Vol. 11048. Lecture Notes in Computer Science. Springer, pp. 49–67 (cited on p. 65).

Winikoff, Michael (2007). "Implementing commitment-based interactions". In: *6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007), Honolulu, Hawaii, USA, May 14-18, 2007*. Ed. by Edmund H. Durfee, Makoto Yokoo, Michael N. Huhns, and Onn Shehory. IFAAMAS, p. 128. DOI: 10.1145/1329125.1329283. URL: http://doi.acm.org/10.1145/1329125.1329283 (cited on p. 21).

– (2017). "Trusting Autonomous Agents". Informal Proceedings of the Enginеering Multi-Agent Systems (EMAS) 2017 Workshop. http://apice.unibo.it/xwiki/bin/download/EMAS2017/Proceedings/EMAS%2D2017%2Dinformal%2Dproceedings.pdf (cited on p. 220).

Winikoff, Michael, Wei Liu, and James Harland (2004a). "Enhancing Commitment Machines". In: *Proc. of Declarative Agent Languages and Technologies II, Second International Workshop, DALT 2004, Revised Selected Papers*. Ed. by João Alexandre Leite, Andrea Omicini, Paolo Torroni, and Pinar Yolum. Vol. 3476. LNCS. Springer, pp. 198–220. DOI: 10.1007/11493402_12. URL: https://doi.org/10.1007/11493402_12 (cited on p. 62).

– (2004b). "Enhancing Commitment Machines". In: *Proc. of DALT 2004, Revised Selected Papers*. Vol. 3476. LNCS. Springer, pp. 198–220 (cited on p. 129).

Winikoff, Michael, Nitin Yadav, and Lin Padgham (2018). "A new Hierarchical Agent Protocol Notation". In: *Autonomous Agents and Multi-Agent Systems* 32.1, pp. 59–133. DOI: 10.1007/s10458-017-9373-9. URL: https://doi.org/10.1007/s10458-017-9373-9 (cited on p. 64).

Winikoff, Michael, Lin Padgham, James Harland, and John Thangarajah (2002). "Declarative & Procedural Goals in Intelligent Agent Systems". In: *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25, 2002*. Ed. by Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams. Morgan Kaufmann, pp. 470–481 (cited on p. 21).

Wood, Peter W., Pierre Boulanger, and Raj S. Padwal (2017). "Home Blood Pressure Telemonitoring: Rationale for Use, Required Elements, and Barriers to Implementation in Canada". In: *Canadian Journal of Cardiology* 33.5, pp. 619–625. ISSN: 0828-282X. DOI: https://doi.org/10.1016/j.cjca.2016.12.018. URL: http://www.sciencedirect.com/science/article/pii/S0828282X16311783 (cited on p. 220).

Wooldridge, Michael John (1992). *The Logical Modelling of Computational Multi-Agent Systems* (cited on p. 2).

Wooldridge, Michael and Nicholas R. Jennings (1995). “Intelligent agents: theory and practice”. In: *Knowledge Eng. Review* 10.2, pp. 115–152. DOI: 10.1017/S0269888900008122. URL: https://doi.org/10.1017/S0269888900008122 (cited on p. 2).

Yadav, Nitin, Lin Padgham, and Michael Winikoff (2015a). “A Tool for Defining Agent Protocols in HAPN: (Demonstration)”. In: *Proc. of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. AAMAS ’15. IFAAMAS, pp. 1935–1936 (cited on p. 64).

– (2015b). “A Tool for Defining Agent Protocols in HAPN: (Demonstration)”. In: *Proc. of AAMAS 2015*. ACM, pp. 1935–1936 (cited on p. 129).

Yolum, Pinar (2006). “Towards Design Tools for Protocol Development”. In: *Proc. of Agent Communication II, International Workshops on Agent Communication, AC 2005 and AC 2006, Selected and Revised Papers*. Ed. by Frank Dignum, Rogier M. van Eijk, and Roberto A. Flores. Vol. 3859. LNCS. Springer, pp. 196–210. DOI: 10.1007/978-3-540-68143-4_14. URL: https://doi.org/10.1007/978-3-540-68143-4_14 (cited on p. 63).

Yolum, Pinar and Munindar P. Singh (2001). “Commitment Machines”. In: *Proc. of Intelligent Agents VIII, 8th International Workshop, ATAL 2001, Revised Papers*. Ed. by John-Jules Ch. Meyer and Milind Tambe. Vol. 2333. LNCS. Springer, pp. 235–247. DOI: 10.1007/3-540-45448-9_17. URL: https://doi.org/10.1007/3-540-45448-9_17 (cited on p. 62).

– (2002). “Commitment Machines”. In: *Proc. of ATAL 2001, Revised Papers*. Vol. 2333. Springer, pp. 235–247 (cited on pp. 129, 146).

Yoo, Sung J. Choi, John A. Nyman, Andrea L. Cheville, and Kurt Kroenke (2014). “Cost effectiveness of telecare management for pain and depression in patients with cancer: results from a randomized trial”. In: *General Hospital Psychiatry* 36.6, pp. 599 –606. ISSN: 0163-8343. DOI: https://doi.org/10.1016/j.genhosppsych.2014.07.004. URL: http://www.sciencedirect.com/science/article/pii/S016383431400173X (cited on p. 220).

Yukish, M., E. Peluso, S. Phoha, S. Sircar, J. Licari, A. Ray, and I. Mayk (1994). “Limits of control in designing distributed C^2 experiments under imperfect communications”. In: *Military Communications Conference, 1994. MILCOM ’94*. IEEE (cited on p. 100).

d’Amorim, Marcelo and Grigore Rosu (2005). “Efficient Monitoring of omega-Languages”. In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, pp. 364–378. DOI: 10.1007/11513988_36. URL: https://doi.org/10.1007/11513988_36 (cited on p. 53).