UiO : Department of Informatics
University of Oslo

Proceedings of the Doctoral Symposium of Formal Methods 2015

Bernhard K. Aichernig and Alessandro Rossini (Eds.)

June 22, 2015

ISBN 978-82-7368-410-3
ISSN 0806-3036

Proceedings of the Doctoral Symposium of Formal Methods 2015

Bernhard K. Aichernig and Alessandro Rossini

Oslo, Norway, 22 June 2015

Preface

This volume contains the research abstracts of the Doctoral Symposium which was held in Oslo on 22 June 2015 as part of the 20^{th} International Symposium on Formal Methods (FM 2015). The Doctoral Symposium provides a helpful environment in which selected PhD students can present and discuss their ongoing work, meet other students working on similar topics, and receive helpful advice and feedback from a panel of researchers and academics.

We received 12 submissions of which 10 were selected for presentation at the Doctoral Symposium. Each research abstract was reviewed by at least three members of the Programme Committee. The emphasis of these reviews was on providing useful feedback to the doctoral students, *e.g.*, pointing them to related work.

The symposium invited Dr. Stijn de Gouw as keynote speaker, whose abstract of his talk *Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)* is included in this proceedings.

The panel of the Doctoral Symposium is formed of three members: Bernhard K. Aichernig (Graz University of Technology, Austria), Ana Cavalcanti (University of York, United Kingdom) and Cristina Seceleanu (Mälardalen University, Sweden).

We would like to thank the authors of the submitted research abstracts, the members of the panel, Dr. Stijn de Gouw, and the members of the Programme Committee for their valuable contributions. Special thanks are due to the excellent local organisation by Einar Broch Johnsen and his team at the University of Oslo. Finally, we would like to thank all symposium participants for making this event fruitful and worthwhile.

June 2015

Bernhard K. Aichernig
Alessandro Rossini

Programme Committee

- Bernhard K. Aichernig, Graz University of Technology, Austria (co-chair)
- Ana Cavalcanti, University of York, UK
- Juan De Lara, Universidad Autonoma de Madrid, Spain
- Zinovy Diskin, McMaster University and University of Waterloo, Canada
- Reiner Hähnle, Technical University of Darmstadt, Germany
- Peter Gorm Larsen, Aarhus University, Denmark
- Yang Liu, Nanyang Technological University, Singapore
- Shaoying Liu, Hosei University, Japan
- Cesar Munoz, NASA, USA
- Alessandro Rossini, SINTEF, Norway (co-chair)
- Augusto Sampaio, Federal University of Pernambuco, Brazil
- Cristina Seceleanu, Mälardalen University, Sweden
- Emil Sekerinski, McMaster University, Canada
- Graeme Smith, University of Queensland, Australia
- Meng Sun, Peking University, China
- Elena Troubitsyna, Aabo Akademi, Finland
- Margus Veanes, Microsoft Research, USA
- Uwe Wolter, University of Bergen, Norway
- Huibiao Zhu, East China Normal University, China

Symposium Programme

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

Stijn de Gouw[1,2]

SDL, Amsterdam, Netherlands
CWI, Amsterdam, Netherlands
sgouw@sdl.com

Abstract Abstract. Some of the arguments often invoked against the usage of formal software verification include the following: it is expensive, it is not worthwhile (compared to its cost), it is less effective than bug finding (e.g., by testing, static analysis, or model checking), it does not work for "real" software. We evaluated these arguments by means of a case study in formal verification.
Tim Peters developed the Timsort hybrid sorting algorithm in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort), by Joshua Bloch - the designer of Java Collections who also pointed out that most bi- nary search implementations were broken. TimSort is today used as the default sorting algorithm for Android SDK, Oracle's JDK, OpenJDK, Python, Apache Hadoop and many other languages and frameworks. Given the popularity of these platforms, the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.
Fast forward to 2015. After we had successfully verified Counting and Radix sort implementations in Java (J. Autom. Reasoning 53(2), 129-139) with a formal verification tool called KeY, we were looking for a new challenge. TimSort seemed to be the ideal candidate for several reasons: it is rather complex, widely used, had a bug history but was reported as fixed in Java 8, and was extensively tested. Unfortunately, we were unable to prove its correctness. A closer analysis showed that this was, quite simply, because TimSort was broken and our theoretical considerations finally led us to a path towards finding the bug (interestingly, that bug appears already in the Python implementation). We show how we did it, derive a mechanically verified bug-free version and discuss the reactions of the developer communities involved in the implementation of the Java and Python standard libraries.

Properties of Communicating Controllers for Safe Traffic Manoeuvres (Research Abstract) *

Maike Schwammberger

Department of Computing Science, University of Oldenburg, Germany
schwammberger@informatik.uni-oldenburg.de

Abstract. This research abstract covers an approach of handling traffic safety in urban traffic scenarios. A two-dimensional interval logic with an underlying complex abstract model is briefly introduced as well as a controller for crossing manoeuvres which uses this logic. As this controller, as well as another controller for traffic manoeuvres for country roads from related work, is based on an idealisation of real-world sensors, the authors first research goal is to extend these controllers to a more realistic approach. The authors main research goal is to examine safety, liveness and fairness properties for these controllers.
Keywords. Multi-dimensional spatial logic, urban traffic, autonomous cars, collision freedom, extended timed automata.

Traffic safety is a relevant topic as driving assistant systems and anytime soon fully autonomously driving cars are increasingly capturing the market. Hilscher et al. introduced an abstract model for proving traffic safety on freeways [1] and country roads [2]. To that end, the authors introduced the two-dimensional interval logic [3] *Multi-lane spatial logic* (MLSL) capable of describing traffic situations. Furthermore they presented an informal concept for lane-change controllers for freeways and country roads.

Martin Hilscher and myself recently extended this abstract model and the logic MLSL to a complex model with intersections for urban traffic scenarios with an underlying graph topology [4]. We introduce semantics of an extension of timed automata [5] to formally substantiate the controllers in [1] and [2]. Furthermore we were able to construct a controller for crossing manoeuvres by adapting the lane change controller from [2] for our purposes. As my current and future work is based on it, I briefly outline some topics of this recent work in the following passages.

Several lines of research exist on traffic safety for autonomously driving cars in different traffic scenarios. For example, successful approaches for automated highway systems with car platoons were presented by the California PATH project [6] and the European Project SARTRE [7]. Another example is

* This research was partially supported by the German Research Council (DFG) in the Transregional Collaborative Research Center SFB/TR 14 AVACS.

the DARPA Urban Challenge 2007 finalist AnnyWAY [8] whose safe algorithm handles moving traffic in urban traffic scenarios.

Among others, an approach to model traffic flow in urban or multi-lane traffic scenarios are *traffic cellular automata*, where macroscopic behaviour is reproduced by a minimal description of microscopic interactions [9]. Another research topic with versatile applications are *intelligent transportation systems*, ranging from car navigation systems to complex traffic management and advanced public transportation systems [10].

Recent Work. Our meaning of *traffic safety* is *collision freedom*. For every car we consider a *safety envelope* to express the space it occupies on the road. This envelope subsumes the cars physical size and braking distance for emergency brakes. We furthermore include the whole width of the lane the car drives on in the safety envelope, to allow a car to slightly move lateral within its lane. With these assumptions, collision freedom means that the safety envelopes of all cars are disjoint at any point in time. In the following passages we assume an idealisation of real-world sensors we call *perfect knowledge*, where every car perceives the whole safety envelope of every other car.

The main idea of our approach is to use purely spatial reasoning, detached from the underlying car dynamics, to prove safety of our controller. Our high-level controllers define a protocol for complex lane change and crossing manoeuvres. A suitable hybrid automaton [11] like controller on a lower dynamical level only needs to manage that the car drives in its safety envelope with the correct speed. An example for an approach to fulfil the task on the dynamical level are the velocity and the steering controller in [12].

Our abstract model (see Fig. 1) is focused on modelling traffic situations at intersections and contains an arbitrary but finite number of discrete *crossing segments* $(c_0, \ldots, c_n)$ and continuous *lanes* $(1, \ldots, n)$. We assume every lane to have one driving direction, but cars might temporarily drive in the opposite direction to perform an overtaking manoeuvre. Every car has a unique identifier $(A, B, C, \ldots)$ and a real value for its position *pos*.

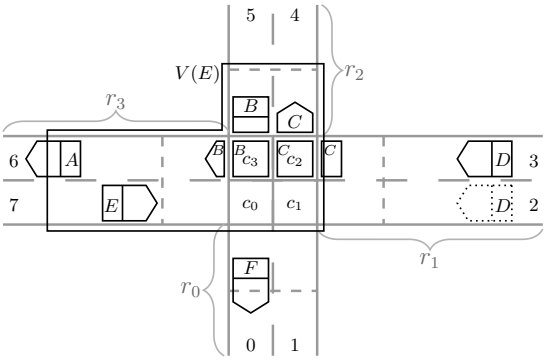

Fig. 1. View $V(E)$ of car E covers the intersection and a finite parts of lanes 6, 7 and 4, 5.

We consider only local parts of the abstract model, where a car E has its own *view* $V(E)$. We distinguish between a *reserved* $(re(E))$ and a *claimed* $(cl(E))$ space on a lane or crossing segment, where a *reservation* is the space the car is driving on and a *claim* is comparable to setting a direction indicator.

If a car intends to turn left or right at an intersection, the view of a car is no longer straight (cf. view $V(E)$ in Fig. 1). Hence, we just unfold the view to a straight *virtual lane* (cf. Fig. 2). A virtual lane only includes those parts of safety envelopes, which are visible within the view under consideration. Concerning Fig. 1, this implies that the part of car C on lane 3 is not contained in the virtual lane in Fig. 2, because it is not part of $V(E)$.

The benefit of unfolding views to virtual lanes is, that on straight lanes we are able to express formulas of our spatial *Urban Multi-Lane Spatial Logic* (UMLSL) which is an extension of previously mentioned MLSL.

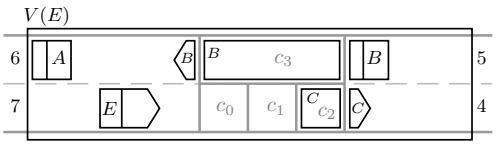

Fig. 2. Virtual lane for view $V(E)$.

In view $V(E)$ depicted in Fig. 1 the UMLSL formula $\langle\phi\rangle \equiv \langle re(E) \frown free \frown re(C)\rangle$ holds. Here $\frown$ is a *horizontal chop operator* comparable to its equivalent in interval temporal logic, that separates adjacent elements in an interval. Furthermore, $free$ describes free space on a lane or crossing segment and $\langle\phi\rangle$ means that ϕ holds *somewhere* in the view.

The topology of our abstract model is given by an *urban road network* (see Fig. 3) where we can express the way a car travels by a *path* assigned to every car. In these networks, intersections are given by strongly connected components of crossing segments that are adjacent to each other (here: c_0, c_1, c_2 and c_3).

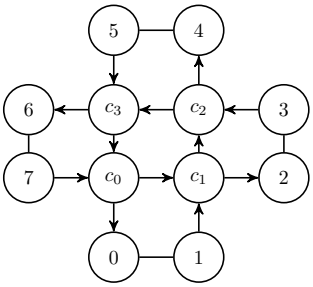

Fig. 3. Road network for Fig. 1.

To actually perform traffic manoeuvres on intersections, we define a *crossing controller*, which is shown in Fig. 4. This controller is an extended timed automaton, where the extensions involve UMLSL formulas as guards and invariants, data variables for cars and lanes and *controller actions* to reserve or claim space on crossing segments. We use the special constant *ego*, to indicate the car E under consideration that owns the view.

The controller action $c_c(ego)$ is used to declare a claim for crossing segments and $r_c(ego)$ to reserve previously claimed crossing segments. For an arbitrary car E crossing segments are claimed, resp. reserved, according to its path. The actions $wd\ c_c(ego)$ and $wd\ r_c(ego)$ are used to withdraw a claim or reservation for a crossing segment, e.g. when the crossing manoeuvre is finished.

To roughly understand this controllers' behaviour, consider for example state q_2: In this state, some crossing segments are already claimed for the car E, according to its path. Instead of the intuitive idea that a claim for lane change manoeuvres can be compared to setting the turn signal, a claim for crossing segments can be understood as an announcement of car E, that it intends to reserve the claimed crossing segments at a later point in time.

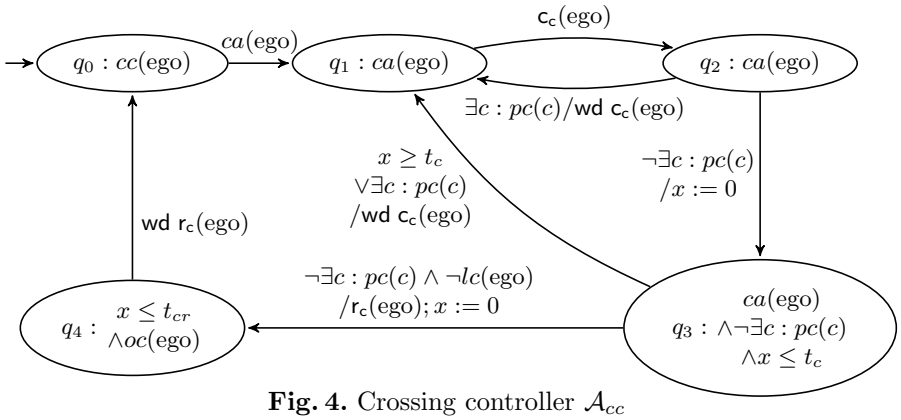

Fig. 4. Crossing controller $\mathcal{A}_{cc}$

While in state q_2 the controller examines if a *potential collision* occurs, by checking if the claim of E intersects with either the claim or reservation of any other car. This is expressed by the formula $\exists c : pc(c) \equiv \exists c : c \neq \text{ego} \wedge \langle cl(\text{ego}) \wedge (re(c) \vee cl(c))\rangle$. If such a potential collision exists, the controller changes back to state q_1 and withdraws its previous claim with the controller action $\mathsf{wd\ c_c}(\mathsf{ego})$. If no potential collision is detected, the controller passes over to state q_3 and might finally be able to reserve his claim and cross the intersection.

Future Work. For both country roads [2] and our recent approach to urban traffic manoeuvres [4] only a concept of *perfect knowledge* was considered, where every car perceives the full safety envelope (comprising the physical size and braking distance) of every other car. I plan to extend this work by considering a more realistic model, where a car's sensors only perceive the physical size of other cars but not their braking distance.

To construct safe controllers in this case, I intend to take the knowledge of other cars into account to increase the range of information on traffic situations available to the car under consideration. To that end, cars need to communicate with each other to prevent collisions, which I will implement by extending the existing controllers by a concept of communication with broadcast channels.

Another interesting, and up to now unconsidered, point, is to examine *liveness properties* of the existing controllers. Do we have a guarantee that on freeways [1] and country roads [2] a car that intends to change a lane will be able to do so finally? For urban traffic scenarios one could even widen the idea of liveness to *fairness properties* to ensure that the time a car waits in front of an intersection does not exceed a certain upper bound.

To express such liveness and fairness properties, I first need to extend the logic MLSL with some time constraints of metric temporal logic [13]. To prove liveness, one could use a tool-driven approach to test reachability of specific desirable states (cf. UPPAAL [14]). In case of the crossing controller for perfect knowledge depicted in Fig. 4, a desirable state would be q_4 because the abbreviation $oc(\text{ego}) \equiv \langle re(\text{ego}) \wedge cs\rangle$ states that the car drives on a crossing segment.

Fairness could be reached by implementing a decentralised scheduling system which grants access to crossing segments according to the *crossing priority* of a car. This priority could increase the longer a car is forced to wait in front of an intersection.

The overall goal of my approach is to adjust the controller for freeways and design new communicating controllers for country roads and crossing manoeuvres, all three without perfect knowledge, that are safe, live and fair.

References

1. Hilscher, M., Linker, S., Olderog, E.-R., Ravn, A.: An abstract model for proving safety of multi-lane traffic manoeuvres. In Qin, S., Qiu, Z., eds.: Int'l Conf. on Formal Engineering Methods (ICFEM). Volume 6991 of LNCS., Springer-Verlag (2011)
2. Hilscher, M., Linker, S., Olderog, E.-R.: Proving safety of traffic manoeuvres on country roads. In Liu, Z., Woodcock, J., Zhu, H., eds.: Theories of Programming and Formal Methods. Volume 8051 of LNCS. Springer (2013) 196–212
3. Moszkowski, B.: A temporal logic for multilevel reasoning about hardware. Computer **18** (1985) 10–19
4. Hilscher, M., Schwammberger, M.: Extending an abstract model for proving safety of motorway manoeuvres to urban traffic scenarios. Manuscript (2015)
5. Alur, R., Dill, D.: A theory of timed automata. TCS **126** (1994) 183 – 2350
6. Lygeros, J., Godbole, D., Sastry, S.: Verified hybrid controllers for automated vehicles. IEEE Transactions on Automatic Control **43** (1998) 522–539
7. Chan, E. et al.: SAfe Road TRains for the Environment (SARTRE): Project final report. Technical report, SARTRE collaborative project (2012)
8. Werling, M., Gindele, T., Jagszent, D., Gröll, L.: A robust algorithm for handling moving traffic in urban scenarios. In: Proc. IEEE Intelligent Vehicles Symposium, Eindhoven, The Netherlands (2008) 168–173
9. Maerivoet, S., De Moor, B.: Cellular automata models of road traffic. Physics Reports **419** (2005) 1–64
10. Figueiredo, L., Jesus, I., Machado, J., Ferreira, J., de Carvalho, J.M.: Towards the development of intelligent transportation systems. In: Intelligent Transportation Systems. Volume 88. (2001) 1206–1211
11. Henzinger, T.A.: The theory of hybrid automata, IEEE Computer Society Press (1996) 278–292
12. Damm, W., Möhlmann, E., Rakow, A.: Component based design of hybrid systems: a case study on concurrency and coupling. In: 17th International Conference on Hybrid Systems: Computation and Control (HSCC'14), Berlin. (2014) 145–150
13. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems **2** (1990) 255–299
14. Behrmann, G., David, A., Larsen, K.: A tutorial on UPPAAL. In Bernardo, M., Corradini, F., eds.: Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004, Springer–Verlag (2004) 200–236

Real-time systems modelling with UML state machines and coloured Petri nets

Mohamed Mahdi Benmoussa

Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, Villetaneuse, France
mahdi.benmoussa@lipn.univ-paris13.fr

Abstract. In this Ph.D. work we deal with the modelling of real-time systems. In a first part, we propose a new method to model UML state machines starting from a text description of real-time systems. In a second part, we propose a set of constructs to express time constraints and annotate the UML state machine model. Since UML does not have an official formally described semantics (a standard semantics approved by OMG), we intend to translate UML state machines with time into (timed) coloured Petri nets. Finally we will implement our algorithms with an automated support to have an automatic translation. As a longer-term work, we would like to prove the correctness of our algorithms by proving an equivalence between the source UML state machine model (e.g. using an operational semantics) and the resulting (timed) coloured Petri net model.

1 The Problem Addressed and its Relevance

To ensure the safety and the correctness of real-time systems, it is necessary to use formal methods. One way is to model systems using a modelling language and then apply formal verification methods. UML is one of the most used modelling languages for its large number of constructs and the diversity of its diagrams, e.g. UML state machine diagrams. UML state machine diagrams are quite expressive and allow the modelling of dynamic behaviours with a rich graphical representation. However, UML does not have an official formally described semantics and the set of syntactic elements to represent time constraints is limited. We propose in this Ph.D. work a new way to specify real-time systems using UML state machines. The result of the specification (the UML state machine model) will be translated into coloured Petri nets. We chose coloured Petri nets because they are a very expressive graphical language with a formal semantics and they allow to perform simulation (using for example CPN Tools [19]).

2 Related Works

Many formalisms have been proposed to handle real-time systems. An approach is the use of UML statecharts [11] to model systems and propose a semantics or a translation to a formal model. The approach proposed in [10] provides a

translation from statecharts (with time extension) to extended hierarchical timed automata. The first part of this approach consists in extending the statecharts model with time to support the modelling of real-time systems. The second part consists in proposing an extended hierarchical timed automaton. However, some syntactic elements taken into account in UML state machines are not available in statecharts [9].

Another approach is an extension for UML called MARTE (Modeling and Analysis of Real-Time and Embedded Systems) to take into account time requirements with a rich syntax. In [3,1] C. André et al. present the time subprofile of MARTE (an OMG UML profile) and the different types of time constraints that this profile allows to express. To verify the properties of time constraints, this approach uses a formal specification language, CCSL (Clock Constraint Specification Language) [17], that is a language to specify clock constraints ([18] proposes a transformation from MARTE/CCSL to Timed Automata for verification). However, we think that due to the richness of the language it is difficult to analyse constraints.

Another approach is to use UML collaborations for time annotations together with UML state machines. Knapp et al. [13] present a tool (HUGO/RT) to verify automatically timed UML state machines models. This tool takes as input UML state machines time-annotated by UML collaborations. The timed state machines are compiled into timed automata that exchange signals and operations via a network automaton. However, the approach has some limitations: the expressiveness of time using UML collaborations is limited, and it is necessary to optimize the number of clocks.

Another way to handle real-time systems is to use patterns based on time constraints (e.g. [16,2]). Mekki et al. [16] use patterns observers of UML state machines to represent temporal requirements, and then translate them into timed automata. The problem of this approach is that the set of time properties taken into account does not allow to describe all kinds of systems. É. André [2] defines a set of correctness patterns encoding common properties that are translated to timed automata/CSP, and their verification reduces to reachability problems. The problem of this approach is the limited number of patterns used to express the time properties.

Shartz et al. [14] propose a framework to analyse UML statechart diagram. The framework analyses Petri net models converted from UML statechart diagrams using the transformation proposed in [12]. The proposed transformation takes into account: simple/composite and orthogonal states, local and external transitions, shallow history pseudo-states, final state, initial pseudo-states, etc. However, the transformation does not consider fork and join pseudo-states, behaviours (entry/exit/do behaviours), variables, guard and action on transitions.

Finally, É. André et al. [7] propose a translation from UML state machines to coloured Petri nets. This approach support a set of syntactic elements of UML state machines such as hierarchy, composition, etc. However this approach does not consider concurrency and time constraints.

3 Solution Scheme and Expected Contributions

The work of my Ph.D. is divided into two parts: the first one concerns the modelling of real-time systems; the second one concerns the translation to a formal model for verification. These two parts are divided into five contributions:

Contribution 1 is to propose syntactic constructs to express time constraints together with a semantics. The aim of those constructs is to provide a simple way to represent and analyse time. It will allow designers using the method of contribution 2 to express in better way their systems.

Contribution 2 is to propose a new method to guide and assist designers for the specification of real-time systems using UML state machines with time (we will use our time constructs to annotate UML state machine model). Our work is inspired by work done in [8]. Starting from a textual description of the system we can apply our specification method to generate the corresponding UML state machine model. The advantage of this approach is both a structured method to model systems but also to avoid as much as possible errors in the modelling phase.

Contribution 3 is to define a new translation from UML state machines to coloured Petri nets with time.

Contribution 4 is the implementation of an automated support to automate the translation of UML state machines with time into timed coloured Petri nets. The tool will allow us to apply the translation on a large set of examples and case studies (gathered from the literature). We will use the result of the translation to apply formal verification methods to prove the system properties.

Contribution 5 is to prove the equivalence between the original UML state machine model and the resulting coloured Petri nets for the validation of the translation. There is no formally described semantics for UML state machines defined by the OMG. However, [15] defines an operational semantics for most syntactical constructs of UML state machines; based on an extension of this semantics to time (that remains to be done), we could prove the equivalence between the two formalisms (UML state machines and coloured Petri nets).

4 Work Progression

My thesis began in October 2013 and is expected to end in September 2016.

Contribution 1: we have handled a set of case studies of real-time systems and we have proposed constructs (such as: concurrency, delay, sequence, precedence...) to express time constraints and take into account (in simple way) the most common time requirements in real-time systems. Our constructs take into account some cases not considered by the other works (e.g. the use of time in states). We are working on the expression of those constructs by defining a new syntax together with a semantics based on a defined grammar. This syntax will allow us to annotate UML state machine with time and be able to model real-time systems. We aim at validating our constructs by modelling a large set of real-time systems.

Contribution 2: we extended the work done in [8] by improving the different steps of the specification method. We applied the method on examples to handle different systems (without time) and to improve its different steps. The application of the method on different case studies of real-time systems will allow us to propose a new method that handles with real-time systems.

Contribution 3: we started by improving the work done in [7] and propose new algorithms to take into account more syntactic elements (e.g. concurrency) in the translation of UML state machines into coloured Petri nets. This improved translation covers a large set of case studies (this work was published in [5,4]). The next step will be the proposal of a translation from UML state machines with time into timed coloured Petri nets.

Contribution 4: we implemented in [6] the translation of [7] using a model-to-text tool (Acceleo[1], based on the approach that takes as input the source model/meta-model and generates the corresponding target model). Due to the absence of the coloured Petri nets meta-model, we were not able to use model-to-model tools (based on the approach that takes as input the source model/meta-model and the target meta-model to generate the corresponding target model). We generate with Acceleo an XMI file that corresponds to the syntax used in CPN tools. This experience shows that Acceleo has a limited syntax to implement our algorithms. Consequently we are working on the implementation of our automated support. We will finish the implementation of the tool to support the two translations (without and with time).

Contribution 5: to be done.

References

1. André, C., Mallet, F., Peraldi-Frati, M.A.: A multiform time approach to real-time system modeling; application to an automotive system. In: IEEE Second Int. Symposium on Industrial Embedded Systems. pp. 234–241. IEEE Computer Society (2007)
2. André, É.: Observer patterns for real-time systems. In: Proc. 18th IEEE Int. Conf. on Engineering of Complex Computer Systems. pp. 125–134. IEEE Computer Society (2013)
3. André, C., Mallet, F., De Simone, R.: Time Modeling in MARTE. In: Forum on specification & Design Languages (FDL'07). pp. 268–273. ECSI (2007)
4. André, É., Benmoussa, M.M., Choppy, C.: Formalisation des diagrammes états-transitions UML concurrents. In: Actes de la session posters du 9e Colloque sur la Modélisation des Systèmes Réactifs (MSR'13) (2013)
5. André, É., Benmoussa, M.M., Choppy, C.: Formalising concurrent UML state machines using coloured Petri nets. In: Proc. of the 6th Int. Conf. on Knowledge and Systems Engineering (KSE'14). Advances in Intelligent Systems and Computing, vol. 326, pp. 473–486. Springer (2014)
6. André, É., Benmoussa, M.M., Choppy, C.: Translating UML state machines to coloured Petri nets using Acceleo: A report. In: 3rd Int. Workshop on Engineering Safety and Security Systems (ESSS'14). pp. 1–7. EPTCS 150 (2014)

[1] https://www.eclipse.org/acceleo/

7. André, É., Choppy, C., Klai, K.: Formalizing non-concurrent UML state machines using colored Petri nets. ACM SIGSOFT Software Engineering Notes 37(4), 1–8 (2012)
8. Choppy, C., Reggio, G.: A method for developing UML state machines. In: Proc. of ACM Symposium on Applied Computing (SAC'09). pp. 382–388 (2009)
9. Crane, M.L., Dingel, J.: UML vs. classical vs. Rhapsody statecharts: not all models are created equal. Software and System Modeling 6, 415–435 (2007)
10. Giese, H., Burmester, S.: Real-time statechart semantics. Tech. rep., Lehrstuhl für Softwaretechnik, Universität Paderborn (2003)
11. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
12. Hu, Z., Shatz, S.M.: Explicit modeling of semantics associated with composite states in UML statecharts. Autom. Softw. Eng. 13(4), 423–467 (2006)
13. Knapp, A., Merz, S., Rauh, C.: Model checking - timed UML state machines and collaborations. In: Proc. of the 7th Int. Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'02). LNCS, vol. 2469, pp. 395–416. Springer (2002)
14. Lian, J., Hu, Z., Shatz, S.M.: Simulation-based analysis of UML statechart diagrams: methods and case studies. Software Quality Journal 16(1), 45–78 (2008)
15. Liu, S., Liu, Y., André, É., Choppy, C., Sun, J., Wadhwa, B., Dong, J.S.: A formal semantics for the complete syntax of UML state machines with communications. In: Proc. 10th Int. Conf. on Integrated Formal Methods (iFM'13). LNCS, vol. 7940, pp. 331–346. Springer (2013)
16. Mekki, A., Ghazel, M., Toguyéni, A.: Timed specification patterns for system validation: A railway case study. In: Proceedings of Informatics in Control, Automation and Robotics (ICINCO'11). vol. 89, pp. 121–134 (2011)
17. Peters, J., Wille, R., Drechsler, R.: Generating systemC implementations for clock constraints specified in UML/MARTE CCSL. In: Proc. Int. Conf. on Engineering of Complex Computer Systems (ICECCS'14). pp. 116–125. IEEE Computer Society (2014)
18. Suryadevara, J., Seceleanu, C., Mallet, F., Pettersson, P.: Verifying MARTE/CCSL mode behaviors using UPPAAL. In: Software Engineering and Formal Methods. LNCS, vol. 8137, pp. 1–15. Springer (2013)
19. Westergaard, M.: CPN Tools 4: Multi-formalism and extensibility. In: Petri Nets. LNCS, vol. 7927, pp. 400–409. Springer (2013)

Test-Case Generation via Language Inclusion for Non-Deterministic Networks of Timed Automata

Florian Lorber

Institute for Software Technology
Graz University of Technology, Austria
florber@ist.tugraz.at

Abstract. In this report we present the core part of a PhD thesis focusing on *model-based testing of real-time systems*. The presented part centers on timed automata and gives a brief overview on fault-based test case generation, determiniziation of bounded timed automata and planned optimizations to both topics.

Keywords: timed automata · real-time systems · model-based mutation testing · bounded determinization of timed automata

1 Introduction

A lot of systems nowadays, especially in safety-critical areas, have to comply to very strict real-time requirements. Consequently, formal models used for verification of such systems need to capture these timing aspects and thus become increasingly complex. In the presented thesis we focus on timed automata [4], one of the most wide-spread formalisms for specifying real-time systems.

Specification models often leave some freedom to the implementation, and thus contain non-determinism, where the implementation may choose between the different paths. However, timed automata with non-determinism are strictly more expressive than deterministic ones, and some importants problems, as e.g. language inclusion, become undecidable for non-deterministic timed automata.

In the presented thesis, we investigate timed automata in the context of model-based testing. In the first year of the doctoral school we already developed, implemented and published an algorithm for fault-based test case generation from deterministic timed automata using language inclusion.

In the second year we investigated removing silent transitions and determinizing timed automata, by restricting ourselves to the bounded case. An according algorithm and its implementation are already developed, but not yet published.

In the last year of the thesis we will investigate networks of timed automata, where the internal communications become hidden and thus are removed. We will also investigate ways to tackle efficiency problems, as for instance the state-space explosion caused by the unfolding of the automata and we will evaluate our approach on several industrial case studies.

2 Related Work

Timed automata were introduced in the 1990s by Alur and Dill [4], and have since received a lot of attention, both on their theoretical and their practical aspects. A recent survey by Waez et al. [15] lists forty different tools that use timed automata in the context of code development and verification. The survey identifies eleven classes of timed automata and almost eighty concrete variants belonging to those classes.

Already Alur and Dill [4] showed that non-deterministic timed automata are more expressive than deterministic ones. Bérard et al. [6] showed that timed automata with silent transitions are even more expressive. Some classes of determinizable timed automata were identified by Baier et al.[5]. They also unfold the non-deterministic automaton into a tree, using one clock for each depth. Contrary to our work, they use an infinite tree and focus on those classes of timed automata that can be determinized, while we cut the tree, to be able to determinize all timed automata bounded to finite traces. Bérard et al. [6] showed that silent transitions without clock resets can be removed without changing the language of a timed automaton. They also show how to remove silent transitions with clock resets, if the silent transitions do not lie on directed cycles. Our algorithm can handle silent transitions with and without clock resets, as long as there are no directed *silent* cycles with clock resets.

Model-based test case generation from timed automata has been done in several approaches. Nielsen and Skou [13] proposed a test case generation framework for non-deterministic (but determinizable) timed automata. The tool UPPAAL Tron [12] performs online testing from timed automata. This approach allows non-determinism and is very adaptive, as the system under test and the model are simulated simultaneously. UPPAAL CoVer [9] is a tool from the same family, which allows the coverage based generation of test cases, according to user-defined observers. Wang et al. [16] performed language inclusion between timed automata via an on the fly determinization. They use infinite trees and thus only terminate for determinizable automata.

Model-based test case generation for real-time systems has been performed with various models other than timed automata. Some have been summarized by Nilsson [14], where he separates the methods into methods based on Process Algebra, Finite State Machines, Temporal Logic, Petri Nets and Informal Methods.

3 Fault-based Test Case Generation of Timed Automata

Model-based mutation testing is a combination of model-based test case generation and mutation testing. It takes a specification model and alters it according to a set of predefined fault models, called *mutation operators*. This creates a set of altered, possibly faulty, models called *mutants*. The mutants can be checked for conformance to the original model. If a mutant conforms, the mutation did

not introduce a fault. However, if the conformance was violated, the check returns a trace indicating where the behavior of the mutant violated the specification. This trace can be converted into a test case which can test whether an implementation behaves like the correct specification or like the mutant. We adopted this approach to timed automata [3]: we defined eight mutation operators, modeling general and timing faults, developed an algorithm for a *tioco*-conformance [10] check via language inclusion and implemented the procedure using the SMT-solver Z3 [8]. The implementation is called MoMuT::TA and is part of the MoMuT toolchain[1]. A release of MuMuT::TA is planned within the next few months. The generated test cases are either the concrete timed traces returned from the SMT-solver or partial models from the specification, containing all locations and transitions that are visited along the trace. The test driver uses these partial models to choose the next inputs and assign *pass*, *fail* and *inconclusive* verdicts. The approach is limited to deterministic automata, as non-determinism might lead to spurious counter examples during the language inclusion. This approach was also only intended for single timed automata. One of the co-authors of the paper later extended the approach to networks of deterministic timed automata [7], by applying the language inclusion check to one single automaton, and then extending the produced test case according to the remaining automata of the network. We also adopted the approach to debugging [1], where we select minimal sets of model-mutants that show the same behavior as a faulty implementation.

4 Determinization of Timed Automata

The limitation of our testing approach to deterministic systems made it inapplicable for many industrial case studies. This problem was hard to overcome, as in general, timed automata can not be determinized [4]. We tackled this by bounding the length of the traces in the automaton, and thus retrieving a tree-shaped determinizable class of timed automata. In the context of testing we are interested in finite traces to reveal faults, thus the bound does not restrict us. The whole approach covers both, the removal of silent transitions and the determinization of the automaton and is integrated in MoMuT::TA. It was not yet published, but detailled information can be found in our technical report [11].

The unfolding during the determinization causes an exponential state-space explosion. Our next step towards applying our approach to real industrial examples will thus be various optimizations. In a recent short paper [2] we already presented a sketch of our ideas. We plan on pruning the unfolded tree during its creation, to create partial models that do not cover all inputs, but still specify the complete output behavior for the remaining part. The pruning can be done according to several different heuristics, ranging from random selection to user-defined distribution.

[1] `https://momut.org//`

5 Conclusion and Future Work

In this report we summarized the current state of our research on timed automata, which is the main aspect of the presented PhD thesis. The doctoral school is supposed to last three years, two of them have passed by now. So far, our research led to eight publications. Three of these publications [1,2,3] dealt with timed automata. At least three more publications on that topic are planned.

In the last year we will work on networks of non-deterministic timed automata. We want to create an unfolded product of all automata, where internal communication channels are hidden and thus removed along with the silent transitions. Then the determinization can be applied as for single automata. Additionally, we want to perform all steps of the approach on the fly.

As soon as this is finished, we will start evaluating the approaches, where we will focus on industrial use cases of our project partners from automotive industry and commonly used timed automata benchmarks.

Acknowledgment

This work is supervised by Prof. Bernhard Aichernig. It has received funding from the ARTEMIS Joint Undertaking under grant agreements N° 269335 and N° 332830 and from the Austrian Research Promotion Agency (FFG) under grant agreements N° 829817 and N° 838498 for the implementation of the projects MBAT, Combined Model-based Analysis and Testing of Embedded Systems and CRYSTAL, Critical System Engineering Acceleration.

References

1. B. K. Aichernig, K. Hörmaier, and F. Lorber. Debugging with timed automata mutations. In *SAFECOMP 2014, Sept. 10-12, 2014. Proceedings*, pages 49–64.
2. B. K. Aichernig and F. Lorber. Towards generation of adaptive test cases from partial models of determinized timed automata. In *A-MOST 2015, In Press.*
3. B. K. Aichernig, F. Lorber, and Dejan Ničković. Time for mutants - model-based mutation testing with timed automata. In *Tests and Proofs*, volume 7942 of *LNCS*, pages 20–38. Springer Berlin Heidelberg, 2013.
4. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
5. Ch. Baier, N. Bertrand, P. Bouyer, and T. Brihaye. When are timed automata determinizable? In *ICALP*, pages 43–54, 2009.
6. B. Bérard, A. Petit, V. Diekert, and P. Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundam. Inf.*, 36(2-3):145–182, November 1998.
7. P. Daca, T. A. Henzinger, W. Krenn, and D. Nickovic. Compositional specifications for ioco testing. In *ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 373–382, 2014.
8. L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

9. A. Hessel and P. Pettersson. Cover-a test-case generation tool for timed systems. *Testing of Software and Communicating Systems*, pages 31–34, 2007.
10. M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.
11. F. Lorber, A. Rosenmann, D. Ničković, and B. K. Aichernig. Bounded determinization of timed automata with silent transitions. Technical Report IST-MBT-2015-01, Graz University of Technology, Institute for Software Technology, 2015. Online. `https://online.tugraz.at/tug_online/voe_main2./getVollText?pDocumentNr=1101473&pCurrPk=83975`.
12. M. Mikucionis, B. Nielsen, and K. G. Larsen. Real-time system testing on-the-fly. In Kaisa Sere and Marina Waldén, editors, *NWPT 2003*, number 34 in B, pages 36–38. Abo Akademi, Department of Computer Science, Finland.
13. B. Nielsen and A. Skou. Automated test generation from timed automata. In *TACAS 2001, held as Part of ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, pages 343–357, 2001.
14. R. Nilsson. Automated selective test case generation methods for real-time systems. Master's thesis, University of Skövde, Department of Computer Science, 2000.
15. Md T. B. Waez, J. Dingel, and K. Rudie. A survey of timed automata for the development of real-time systems. *Computer Science Review*, 9:1–26, August 2013.
16. T. Wang, J. Sun, Y. Liu, X. Wang, and S. Li. Are timed automata bad for a specification language? language inclusion checking for timed automata. In *TACAS 2014, Held as Part of ETAPS 2014*, pages 310–325, 2014.

Trace-length Independent Runtime Monitoring

Xiaoning Du

School of Computer Engineering, Nanyang Technological University
xndu@ntu.edu.sg

1 Introduction

Aircraft flight control, medical devices and nuclear systems are prime examples of safety-critical Cyber-Physical Systems (CPS), whose failure could result in loss of life, significant property damage, or damage to the environment. Generally, systems that can cause information or financial loss are also considered as safety-critical [15], such as smartphones. Mainly, the functions in such systems are controlled by embedded computers and it is imperative to verify the functional correctness of the embedded software [10]. Traditional verification techniques such as model checking often suffer from practical infeasibility and/or theoretical impossibility [18]. As an alternative to verification and off-line testing, *runtime monitoring* (or runtime verification) is proposed to verify system properties at execution time by using an online algorithm to check whether a system trace satisfies a temporal property [3]. It is a novel light-weight verification technique, where properties are to be checked against the executing system rather than the abstracted model which is sometimes hard or impossible to establish. Practically, runtime verification on these platforms is always with tight timing constraints and has high demands for the performance of the monitoring algorithms on both time and space due to limited computing and storage resources of the embedded systems. So it is not feasible to record the entire history of events happening in the past. For our intended applications, once we fix the policies to be monitored, the monitoring algorithms' space requirement and running time at each state are expected to be constant and will not increase as the traces grow.

Before designing the monitoring algorithms, we need to define a proper logic specification language with which to express the security related properties. As far as we know, Linear temporal logic (LTL) has been widely used as a specification language to specify runtime properties of systems and languages. A trace-length independent monitoring algorithm for LTL has already been designed by Havelund *et al.* [14]. Traditionally, the use of LTL is concerned mainly with qualitative properties, such as relative ordering of events, or eventuality of events, etc. According to our survey on Android malware detection, some attack patterns cannot be stated as pure LTL formulas as they require specifications of quantitative measures such as frequency of certain activities (e.g., sending SMS) commonly found in botnet attacks. From the above example we can see that LTL is expressively restricted in the properties it can specify, and some other semantical components need to be added on to broaden its expressiveness. There are several aspects that deserve consideration, such as coping with variables ranging over infinite domains, providing both universal and existential quantification, allowing quantitative temporal operators, or simultaneously handling past and future temporal

operators, as mentioned in [3]. Properties to be specified may take various forms and can be given at varying degrees of abstraction. Aiming at stipulating as many properties as possible, we may achieve an extremely complicated logic, which is hard to be monitored at runtime. Now we are encountered with the classic tradeoff between expressiveness and complexity. An interesting question here is to what extend we can broaden the expressiveness of the logic languages that can be monitored trace-length independently. Therefore, we will conduct a research on this problem, and our approach and expected contributions are as follows.

Overall Approach. Basically, the aspects listed above will be considered and fused to pure LTL, and some restrictions will be set to enforce trace-length independence. Identification of the syntactic restrictions will help to tell from the syntax of the formula whether it can be enforced in a trace-length independent way, but it is not always possible. Then some semantic restrictions will be imported, e.g., in [11] it relies on some external tools to find the equivalence classes etc. Even if extra demands on the monitoring algorithms could be put forward for systems with different storage and computing resource, we will treat the trace-length independent algorithms as an initial approach and see how far we can push this idea on the runtime verification of embedded systems.

Expected Contributions. Above all, an expressive enough formal specification language that can be used to specify a broad class of properties on the resource-limited platforms will be designed. Besides, the corresponding trace-length independent monitoring algorithm will be designed and its correctness and trace-length independence will be proved theoretically. Finally, we will implement the monitoring algorithms on both software and hardware platforms to demonstrate its general usability and make evaluation on its performance.

2 Current Progress of Research

On account of the extremely limited computating and storage resource in embedded systems, it is not practical to record the entire event history of the system. This critical requirement drives us to discover some trace-length independent monitoring algorithms. Although the concept of trace-length independent monitoring was proposed in the year 2013 in [4], there are already some prior works which imply this property in their algorithm designs as in [13,14]. Havelund *et al.* [14] propose a trace-length independent monitor algorithm for past-time LTL (*pt*LTL) in which all logical operators are expressed in recursive form. One needs to maintain only two states for each subformula of a policy to enforce without losing the completeness of the algorithm. This monitoring algorithm does provide a foundation based on which we can explore trace-length independent monitoring algorithms for more complicated logic specification languages with more powerful expressiveness. Recent developments in metric LTL and its extensions with aggregate operators allow some quantitative properties to be specified, augmenting the expressiveness of the logics.

Inspired by the aggregation operators in database query language like SQL, Basin *et al.* [2] extend metric first-order temporal logic (MFOTL) with aggregation operators,

like SUM, CNT, MAX and AVG, and proposed a monitoring algorithm for language. The core of this work is to translate policies specified with the extended MFOTL to the corresponding extended relational algebra. For their monitoring algorithm, functions are handled similarly to Prolog. Even through some optimizations are taken to accelerate computations in monitoring, the aggregation operators are out of their consideration. Another language, SOLOIST [8], is based on a many-sorted first-order metric temporal logic and extended with new temporal modalities that support aggregate operators for events occurring in a certain time window. For its monitoring, Bianculli *et al.* [9] proposed to translate the formulae in SOLOIST to formulae of CLTLB($\mathscr{D}$) [7], and Bersani *et al.* [6] presented an approach to encode SOLOIST formulae into QF-EUFIDL formulae. Nevertheless, both approaches depend on SMT-solver to do the final satisfiability checking. The evaluations of the above two works show that increasing time and memory will be needed when the length of the trace grows.

Up until now, there has been so far no study on trace-length independence monitoring for LTL with aggregate operators like the counting quantifier. We solved this problem in [11], and a policy specification language was proposed based on a past-time variant of LTL, extended with an aggregate operator called *counting quantifier* to specify policies checking how many times some sub-policies are satisfied in the past. We show that a broad class of policies, but not all policies, specified with our language can be monitored in a trace-length independent way, and provide a concrete algorithm to do so. We also implement and test our algorithm in an existing Android monitoring framework and showed that our approach can effectively specify and enforce quantitative policies drawn from real-world Android malware studies. This is a good start for us to go further in the way of increasing the expressiveness of logics originating from LTL.

3 Future Work

Metric Temporal Operators. Pure *pt*LTL cannot specify temporal constraints on logic operators. Real-time runtime verification, however, prunes to specify some time related properties. Imperative requirements for the temporal logic operators are put forward in [1,16,21] on analysis of intrusion detection. Coincidentally, the essential parts of all of their approaches to detecting potential attacks need to count how many times a sub-policy is satisfied within some previous time intervals, e.g., in detection of *TCP syn flood attack*, it is required to check how many times the sender fails to respond a package with *ack* flag in the previous time unit. Thati *et al.* [20] design a monitoring algorithm for Metric Temporal Logic (MTL). When restricted to the past time fragment, the monitoring algorithm turns out to be trace-length independent, benefiting from the recursive definitions of the semantics of operators. We will try to adapt our current logic to include a metric temporal counting quantifier that counts how many times a sub-policy is satisfied during some past time periods.

First-order Quantifiers and Recursive Definition. Gunadi *et al.* [13] lead up the idea to extend past-time MTL (*pt*MTL) with first-order quantifier and recursive definition, providing a solution to monitoring the *privilege escalation* on Android. They also present

a trace-length independent algorithm for monitoring properties written in their specification language under various fine grained constraints. However, they do not deal with quantifiers directly in their algorithm. Instead, the quantifiers are expanded into pure propositional connectives, which are exponential in the number of variables in the policy. Another study [5] investigating the monitoring problem of first-order temporal logic gives a monitoring algorithm using *spawning automata*, which is in principle trace-length independent. It is worth to be investigated whether techniques using spawning automata can be adapted to the setting in [13] to allow a *lazy* expansion of quantifiers as needed. It is not possible to design trace-length independent monitoring algorithm in the unrestricted first-order LTL, so the challenge is to find a suitable restriction that can be enforced efficiently.

Future Temporal Operators. Up until now, we just consider the past time fragment of linear temporal logics. In practice, however, future time semantics are indispensable. For example, under some situations, it is required that a user who has a login must logout within 3 hours eventually. In [12] a monitor circuit has been implemented to avoid the exponential space consumption of bounded-future subformulas.

Implementation on Hardware. Classic runtime verification is to instrument the code base of the system, while this kind of instrumentation cannot always be applicable to some embedded systems due to the non-instrumentable hardware or mechanical parts. Even if the instrumentation is applicable, additional verification overhead may alter timing behavior and memory consumption. It can be seen that runtime verification on hardware makes excessive demands on the efficiency of the monitoring algorithms on both time and space. We will try to port out monitor on FPGA platforms similar to [19,12,17]. In [17], Reinbacher *et al.* present an on-line algorithm to check a *pt*MTL formula on executions with discrete time domain. They also discuss a reconfigurable hardware realization of their observer algorithm that provides sufficient flexibility to allow for changes of formulas without necessarily re-synthesizing the hardware observer. Taking advantage of the highly-parallel nature of hardware designs makes their observer algorithms much efficient. The resulting hardware blocks can be applied in prototyping and runtime verification of embedded real-time systems. We will follow a similar line to provide complete implementation of our trace-length independent monitoring algorithm on hardware.

Acknowledgment. We thank the anonymous referees for their helpful comments. This research is supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate. My supervisors Yang Liu and Alwen Tiu have provided relevant insights related to this work.

References

1. A. Ahmed, A. Lisitsa, and C. Dixon. A misuse-based network intrusion detection system using temporal logic and stream processing. In *NSS*, pages 1–8, 2011.

2. D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring of temporal first-order properties with aggregations. In *RV*, pages 40–58, 2013.
3. D. A. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *FSTTCS*, pages 49–60, 2008.
4. A. Bauer, R. Goré, and A. Tiu. A first-order policy language for history-based transaction monitoring. In *ICTAC*, pages 96–111. 2009.
5. A. Bauer, J.-C. Küster, and G. Vegliach. From propositional to first-order monitoring. In *RV*, pages 59–75, 2013.
6. M. M. Bersani, D. Bianculli, C. Ghezzi, S. Krstić, and P. San Pietro. Smt-based checking of soloist over sparse traces. In *FASE*, pages 276–290. 2014.
7. M. M. Bersani, A. Frigeri, A. Morzenti, M. Pradella, M. Rossi, and P. S. Pietro. Constraint ltl satisfiability checking without automata. *Journal of Applied Logic*, 12(4):522 – 557, 2014.
8. D. Bianculli, C. Ghezzi, and P. San Pietro. The tale of soloist: a specification language for service compositions interactions. In *Formal Aspects of Component Software*, pages 55–72. 2013.
9. D. Bianculli, S. Krstic, C. Ghezzi, and P. San Pietro. From soloist to cltlb (d): Checking quantitative properties of service-based applications. 2013.
10. A. Dokhanchi, B. Hoxha, and G. Fainekos. On-line monitoring for temporal logic robustness. In *RV*, pages 231–246, 2014.
11. X. Du, Y. Liu, and A. Tiu. Trace-length independent runtime monitoring of quantitative policies in ltl. In *FM*, 2015.
12. B. Finkbeiner and L. Kuhtz. Monitor circuits for ltl with bounded and unbounded future. In *RV*, pages 60–75, 2009.
13. H. Gunadi and A. Tiu. Efficient runtime monitoring with metric temporal logic: A case study in the android operating system. In *FM*, pages 296–311. 2014.
14. K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *TACAS*, pages 342–356, 2002.
15. J. C. Knight. Safety critical systems: challenges and directions. In *ICSE*, pages 547–550, 2002.
16. P. Naldurg, K. Sen, and P. Thati. A temporal logic based framework for intrusion detection. In *FORTE*, pages 359–376. 2004.
17. T. Reinbacher, M. Függer, and J. Brauer. Real-time runtime verification on chip. In *RV*, pages 110–125, 2013.
18. T. Reinbacher, M. Függer, and J. Brauer. Runtime verification of embedded real-time systems. *Formal Methods in System Design*, 44(3):203–239, 2014.
19. T. Reinbacher, K. Y. Rozier, and J. Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *TACAS*, pages 357–372. 2014.
20. P. Thati and G. Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.
21. J. Viinikka and H. Debar. Monitoring ids background noise using ewma control charts and alert information. In *Recent Advances in Intrusion Detection*, pages 166–187, 2004.

Inheritance and refinement of trustworthy component-based systems

José Dihego

Universidade Federal de Pernambuco (UFPE) - Recife, PE, Brazil
jdso@cin.ufpe.br

Abstract. We propose inheritance and refinement relations for a CSP-based component model (BRIC), which supports a constructive design based on composition rules that preserve behavioural properties such as deadlock freedom. The inheritance relations allow extension of functionality, whilst preserving service conformance, which we define by means of a substitutability test. We also establish an algebraic connection between inheritance and refinement and outline how they can be verified in the FDR modelchecker.

1 Introduction

Component-based model driven development (CB-MDD) [8] is a well recognised approach to develop complex systems and has been successfully applied in industry. The $\mathcal{BRIC}$ component model [6] is a formal approach to CB-MDD: it defines components and compositions, where behavioural properties are ensured by construction. Nevertheless, $\mathcal{BRIC}$ does not have notions for component refinement and inheritance, an essential condition to evolve specifications reliably.

Different related work have tried to develop formal foundations for CB-MDD [6, 3, 1], but they lack to offer refinement and inheritance in a trustworthy stepwise development discipline. This is the main objective of this work and, as far as the author is aware, the first attempt to integrate inheritance/refinement into a formal CB-MDD approach where behavioural properties emerge by construction.

Section 2 presents our inheritance and refinement relations for $\mathcal{BRIC}$ based on a novel concept called behavioural *convergence*. We also contextualise our contribution in the light of relevant related work. Section 3 presents our conclusions and the expected results.

2 Context and progress

A component in $\mathcal{BRIC}$ is defined as a contract that specifies its behaviour (a CSP [7] process), communication points (CSP channels) and their types. Formally, a component contract $Ctr : \langle \mathcal{B}, \mathcal{R}, \mathcal{I}, \mathcal{C} \rangle$ comprises an observational behaviour $\mathcal{B}$ specified as a CSP process, a set of communication channels $\mathcal{C}$, a set of interfaces $\mathcal{I}$ and a **total** function $\mathcal{R} : \mathcal{C} \to \mathcal{I}$ between channels and interfaces of the contract. We require the CSP process $\mathcal{B}$ to be an I/O process, which is a non-divergent processes with infinite traces. Moreover, it offers the environment the choice over its inputs (external choice) but reserves the right to choose between its outputs (internal choice). It is suitable for a widely range of specifications, including the

client-server protocol, where a client sends requests (inputs) to a server that decides internally how to respond (output).

Contracts can be composed using any of the four rules available in the model: interleaving, communication, feedback, or reflexive (Figure 1). Each of these rules impose different side conditions, which must be satisfied by the contracts and channels involved in the composition in order to guarantee deadlock freedom by construction. Interleave composition creates a loosely coupled component by putting together two others, which communicate independently with the environment. The communication composition connects a pair of channels of two different components, where one inputs the outputs generated by the other. Self connections are possible by the use of reflexive/feedback compositions: reflexive is more general than feedback, as it does not constrain the channels to be connected, whereas feedback composition impose conditions to allow local analysis.

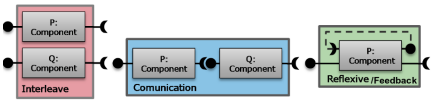

Fig. 1. Composition rules

2.1 Component inheritance and refinement

The concept of inheritance in the object-oriented paradigm is well-established [5]. A strong form of inheritance is subtyping [4], which allows reuse and extensibility and, moreover, fulfils the principle of type substitutability [9]: an instance of the subtype should be usable wherever an instance of the supertype was expected, without a client being able to tell the difference. Recently, efforts have been made to extend this concept to process algebras as CSP. Notably, in [10] the author proposes four types of behavioural inheritance relations to Labelled Transition Systems. Although very promising, these relations do not consider specifications that distinguish inputs from outputs, such as $\mathcal{BRIC}$. We base our approach towards an inheritance relation for $\mathcal{BRIC}$ on the concept of *convergence* [2]: a convergent process is allowed to do the same as or enable more inputs than its parent process, but is restricted to do the same or less outputs in *convergent points*. A convergent point represents a state reachable by both the original and the convergent process when doing two convergent sequences of events: these sequences differ only because the convergent process is allowed to do extra inputs, i.e. new-in-context inputs not allowed by the original process at related states.

An I/O process T' is convergent to T (T' `io_cvg` T, Definition 1) if in each converging point of their execution it can offer more or equal inputs but is restricted to offer less or equal outputs. A convergent I/O process can engage in more inputs to take more deterministic decisions on what to output. In Definition 1, $\mathcal{T}(T)$ and $\mathcal{F}(T)$ stand for the traces and failures of a process T, respectively; Σ stands for the alphabet of all possible events, Σ^* is the set of possible sequences of events from Σ, the input events are contained in Σ ($inputs \subseteq \Sigma$) and $in(T, t)$ is a function that yields the set of input events that can be communicated by the I/O process T after some $t \in \mathcal{T}(T)$, therefore $in : I/OProcess \times \Sigma^* \to inputs$. Additionally, if $t_1 \leq t_2$, it means that t_1 is a subtrace of t_2.

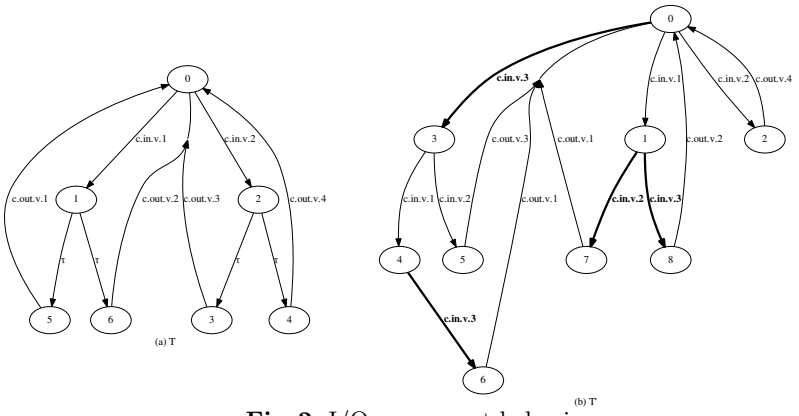

Fig. 2. I/O convergent behaviours

Definition 1 (I/O convergent behaviour). *Consider two I/O process T and T'. T' is an I/O convergent behaviour of T (T' `io_cvg` T) if, and only if:*

$$\forall(t', X) \in \mathcal{F}(T'), \exists(t, Y) \in \mathcal{F}(T) \bullet \begin{pmatrix} t' \texttt{ cvg } t \wedge \\ Y \cap inputs \supseteq X \cap inputs \wedge \\ Y \cap outputs \subseteq X \cap outputs \end{pmatrix} \quad \text{where, } t' \texttt{ cvg } t \Leftrightarrow \left(\begin{array}{l} (\#t' > \#t) \ \wedge \ \exists t_1, t_3 : \Sigma^*, \exists ne : \Sigma \mid \\ \left(\begin{array}{c} t' = t_1 \frown \langle ne \rangle \frown t_3 \ \wedge \ t_1 \leq t \wedge \\ ne \in inputs \wedge ne \notin in(T, t_1) \wedge \\ t_1 \frown t_3 \texttt{ cvg } t \end{array} \right) \end{array} \right) \vee (t' = t)$$

It can be useful to offer other events after a new input and before converging to its original behaviour. This extension to convergence allows convergent processes to add more implementation details decreasing the abstraction level of specifications. An extended convergent process T' can accept any event not expected by T (T' `io_ecvg` T, Definition 2) in an extended convergent point of their execution, provided a new-in-context input has happened, marking the start of the extended convergent behaviour of T'. Figures 2(a) and 2(b) depict two convergent I/O processes, T' `io_cvg` T, where we bold face new in-context input events allowed by this relation. Figures 3(a) and 3(b) shows the case where T' `io_ecvg` T: after a new-in-context input event, T' can communicate everything it wants before converging to T.

Definition 2 (I/O extended convergent behaviour). *Consider two I/O process T and T'. We say that T' is an I/O extended convergent behaviour of T (T' `io_ecvg` T), if and only if:*

$$\forall(t', X) \in \mathcal{F}(T'), \exists(t, Y) \in \mathcal{F}(T) \bullet \left(\begin{array}{c} t' \texttt{ ecvg } t \wedge \\ \left(\begin{pmatrix} Y \cap inputs \supseteq X \cap inputs \wedge \\ Y \cap outputs \subseteq X \cap outputs \end{pmatrix} \right. \\ \left. \vee \ (\Sigma Y \subseteq X) \right) \end{array} \right) \quad \text{where, } t' \texttt{ ecvg } t \Leftrightarrow \left(\begin{array}{l} (\#t' > \#t) \ \wedge \ \exists t_1, t_2, t_3 : \Sigma^*, \exists ne \in \Sigma \mid \\ \left(\begin{array}{c} t' = t_1 \frown \langle ne \rangle \frown t_2 \frown t_3 \wedge t_1 \leq t \wedge \\ ne \in inputs \wedge ne \notin in(T, t_1) \wedge \\ set(t_2) \cap (in(T, t_1) \cup out(T, t_1)) = \emptyset \wedge \\ t_1 \frown t_3 \texttt{ ecvg } t \end{array} \right) \end{array} \right) \vee (t' = t)$$

Component inheritance. Our definition of inheritance deals with component structural and behavioural aspects. Structurally, it guarantees that the inherited component preserves at least its parent's channels and their types. Regarding behaviour, they are related by convergence. Additionally, it guarantees, for the purpose of substitutability, that the inherited component only refines the behaviour exhibited by common channels (default channel congruence) or that

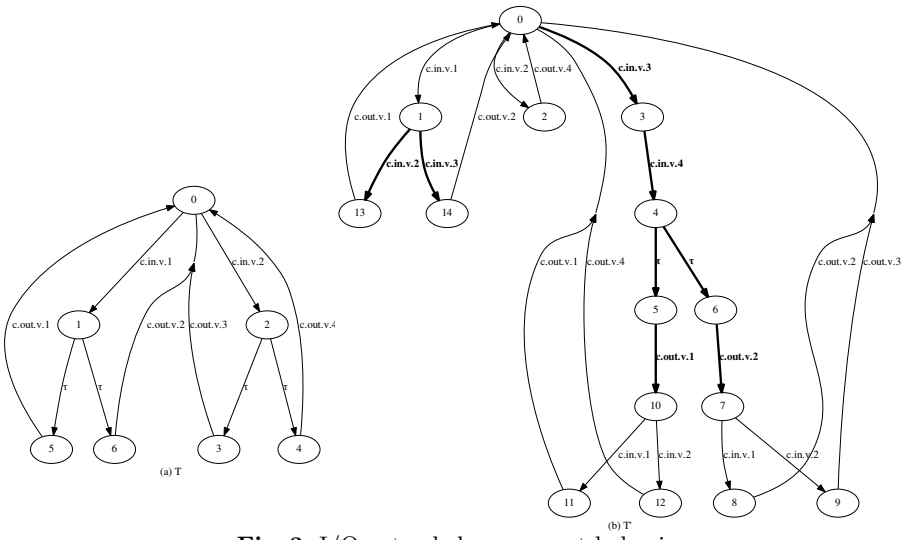

Fig. 3. I/O extended convergent behaviours

additional inputs over common channels are not exercised by any possible client of its parent (input channel congruence). We also contribute with a refinement relation for $\mathcal{BRIC}$, which is based on failures refinement of CSP [7].

Definition 3 (component inheritance). *Consider T and T' two $\mathcal{BRIC}$ components, such that $\mathcal{R}_T \subseteq \mathcal{R}_{T'}$. We say that T' inherits from T:*

- *by convergence:* $T \leftarrow_{cvg} T' \Leftrightarrow \mathcal{B}_{T'}$ `io_cvg` $\mathcal{B}_T$
- *by extended convergence:* $T \leftarrow_{ecvg} T' \Leftrightarrow \mathcal{B}_{T'}$ `io_ecvg` $\mathcal{B}_T$

provided *their corresponding channels are default or input channel congruent.*

Definition 4 (component refinement). *Consider T and T' two $\mathcal{BRIC}$ components. We say that T' refines T ($T \sqsubseteq_{\mathcal{B}} T'$) if and only if:*

$$\mathcal{B}_T \sqsubseteq_{\mathrm{F}} \mathcal{B}_{T'} \wedge \mathcal{R}_T \subseteq \mathcal{R}_{T'}$$

We expect to prove the next proposition as a theorem. It relates refinement and inheritance presenting inheritance as a relation that supports evolution, by which we can always extract an abstraction component from another.

Proposition 1 (inheritance and refinement). *Let T, T_{con} and T_{abs} be component contracts, such that $T_{abs} \leftarrow_{cvg} T_{con}$ or $T_{abs} \leftarrow_{ecvg} T_{con}$, then $T \sqsubseteq_{\mathcal{B}} T_{con}$.*

Component inheritance based on convergence guarantees substitutability in the sense that if $T \leftarrow_{ecvg} T'$ (or $T \leftarrow_{cvg} T'$), then T' can replace T in any composition it can take part without introducing deadlock. Moreover, any component acting as a T client cannot tell the difference when using T', even if its new functionalities are exercised by other clients.

Checking convergence in FDR. An important issue we must address is how to build an automated strategy to verify convergence. Our strategy is to construct a tester process such that: T' `io_cvg` $T \iff Tester\_cvg(T) \sqsubseteq_{\mathrm{F}} T'$, where T and T' are components.

Consider T an I/O process, then $cvg^{+}T$ stands for a set of I/O processes such that: $\forall T' \in cvg^{+}T \mid T'$ `io_cvg` T. The set $cvg^{+}T$ is infinite, which makes this use prohibitive for any implementation that aims to traverse it entirely. A finite subset of $cvg^{+}T$ is given by $cvg^{+^n}T$, which stands for the T convergent processes whose depth differ from that of T at most by n ($n \in \mathbb{N}$). An I/O process depth is given by the greatest number of events that makes a process came back, only once, to its initial state.

Our strategy is to construct $Tester\_cvg(T)$ to be the *lower bound* process of $cvg^{+^n}T$ under failures refinement. Therefore, T' `io_cvg` T is reduced to $Tester\_cvg(T) \sqsubseteq_{\mathrm{F}} T'$. The same reasoning applies to check T' `io_ecvg` T.

3 Conclusions and expected results

This paper presents an overview of our developments towards a theory for component refinement and inheritance for CB-MDD based on behavioural convergence. We expect to create a sound theory to safely evolve CB-MDD specifications and to explore our developments on well-known critical systems by mechanising the crucial aspects of our theory.

Acknowledgments. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES [1]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.

References

1. Zhenbang Chen, Zhiming Liu, Anders P. Ravn, Volker Stolz, and Naijun Zhan. Refinement and verification in component-based model-driven design. *Sci. Comput. Program.*, 74(4):168–196, February 2009.
2. José Dihego, Augusto Sampaio, and Marcel Oliveira. Constructive extensibility of trustworthy component-based systems. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15. ACM, 2015.
3. He Jifeng, Xiaoshan Li, and Zhiming Liu. rcos: A refinement calculus of object systems. *Theoretical Computer Science*, 365:109 – 142, 2006. Formal Methods for Components and Objects Formal Methods for Components and Objects.
4. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
5. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall International, 2nd edition, 1997.
6. R. Ramos, A. Sampaio, and A. Mota. Systematic development of trustworthy component systems. In *2nd World Congress on Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2009.
7. A. W Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
8. Clemens Szyperski. *Component Software: Beyond Object-oriented Programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.
9. Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings of the European Conference on Object-Oriented Programming*, London, UK, 1988. Springer-Verlag.
10. Heike Wehrheim. Behavioral subtyping relations for active objects. *Form. Methods Syst. Des.*, 23(2):143–170, 2003.

[1] www.ines.org.br

Component-based CPS Verification: A Recipe for Reusability

Andreas Müller
Johannes Kepler University Linz, Altenbergerstr. 69, 4040 Linz, Austria
andreas.mueller@jku.at

1 Overview

Cyber-physical systems (CPS) are today pervasively embedded into our lives and increasingly act in close proximity as well as with direct impact to humans. Because of their safety-criticality, we have to ensure *correctness properties*, such as *safety* and *liveness*. Thus, formal verification techniques to analyze CPS are of paramount importance to guarantee these properties.

Formal verification methods rest on *models* capturing the *discrete and continuous dynamics* of a CPS (i. e., hybrid system models), which abstract from implementation details to facilitate verification. Since formal verification of hybrid systems is known to be undecidable for realistic models, current methods make a trade-off between full automation (*model checking* and *reachability analysis* restricted to certain classes of hybrid systems, e. g., [5]) and model expressiveness (*deductive verification* of complex models mixing automated and human guided proofs, e. g., [7]). To make human guidance feasible despite complex continuous dynamics, CPS practically mandate for techniques to reduce system complexity.

We study decomposing a model into smaller components, which can be proven separately before re-composing them to a fully verified model. With current deductive verification techniques for CPS, however, the price for such component-based development is full re-verification on every composition step, which is a nuisance if human guidance is required each time.

Vision: reduce modeling and verification effort and complexity, and increase reusability by component-based CPS development.

Although component-based software engineering in general has seen extensive research, only few approaches explicitly deal with CPS, like Damm et al. [3], who propose a design methodology for hybrid systems based on sequential composition of components using alarms.

A field closely related to component-based verification is assume-guarantee reasoning (AGR), which was originally intended as a device to counteract the state explosion problem in model checking by decomposing a verification task into subtasks. In AGR, individual components are analyzed together with *assumptions* about their context and *guarantees* about their behavior (i. e., a component's "contract"). Benvenuti et al. [1] propose an approach to check these contracts for hybrid automata with non-linear dynamics, using the reachability toolbox Ariadne. AGR is often used along with abstraction/refinement ap-

proaches (e. g., [2]) and the rules are often circular in the sense that one component is verified in the context of the other and vice-versa (e. g., [5]).

Summarizing, current component-based approaches are often limited to linear dynamics (e. g., [2,3]), need to abstract away continuity (e. g., [5]) or rely on reachability analysis, over-approximation and model checking (all of the above). In the next section we will propose a *component-based* modeling and verification approach based on *deductive verification*.

2 Research Approach

We will follow a three-step research approach, where we (i) conduct initial case studies, to gain insight into decomposition options in deductive verification, (ii) develop component-based modeling and verification for hybrid systems and (iii) evaluate our findings with case studies.

For this work, we use *differential dynamic logic* ($\mathsf{d}\mathcal{L}$), which is a first-order dynamic logic that has a notation for hybrid systems as *hybrid programs* and its hybrid deductive verification tool KeYmaera [7] that allows proving correctness properties of these hybrid programs. Hybrid programs allow sequential composition, non-deterministic choice, repetition and assignment, deterministic assignments, state checks and continuous evolution. Here a hybrid interpretation of time is used, where time evolves continuously and without discretization during continuous evolution and in discrete steps otherwise. The hybrid programs can be embedded into $\mathsf{d}\mathcal{L}$ formulas using the modalities $[a]$ and $\langle a \rangle$ to reason about all runs of a hybrid program a or at least one run of a respectively.

The ultimate goal is a framework that provides a set of composition operations that transfer verified properties of the internal and external behavior of components to composites. Users then have to decompose a system into components, verify their internal and external behavior in isolation according to our framework and use the composition operations to recreate the overall system. The properties of the composed system can be derived from its components.

2.1 Initial Compositional Modeling and Verification Case Studies

Based on prior experience with road traffic[1] we will introduce compositional modeling and verification of road networks. Road network capacity analysis involves highly repetitive parts, such as traffic lights or merging roads. On that account, we coarsely approximate traffic flow using linear water tank models. These limitations allow studying component interfaces, continuous dynamics and composition in a restricted setting of a single composition operation (i. e., instantaneous, loss-less passing of flow) and simplified continuous dynamics (i. e., approximate flows with linear ordinary differential equations (ODEs)).

We will complement the macroscopic network flow study, which is highly time-dependent, as flow accumulates over time, with a microscopic case study

[1] Gained in the research project CSI (`http://csi.situation-awareness.net`).

on autonomous cars, which mainly have to deal with their ever changing surroundings and where safety criticality is even more of an issue. Thus, we have to extend our previous approach with multiple composition operations (e. g., noisy measurements) and non-linear continuous dynamics (e. g., curved trajectories).
Status: Traffic Components with Maximum Flow. We modeled three types of flow components (traffic light, two-way-merge, two-way-split) and verified that they will not exceed their capacity for some time T_{local}, when considering the maximum possible in- and outflow of a component. Furthermore, we introduced a composition operation which results in a safe composite, if both components follow their local safety property and a simple arithmetic composition relation holds. This condition has to be checked using designated values when composing components to form the whole system. While deciding the validity of a safety property for the entire traffic network would be doubly exponential in the (assumable large) number of variables [4], the evaluation of the arithmetic condition over the reals for concrete numbers is linear in the formula size [6]. Thus, the arithmetic composition relation can be checked at scale in a modeling tool when building road networks. When it holds, the local safety property transfers to the whole system, ensuring no overflow until a time T_{global}.

2.2 Component-based CPS Development

Component-based CPS development and verification requires definition and verification of components (i. e., their *internal behavior*) and their interfaces (i. e., their *external behavior*), as well as their *verified composition*. The concepts in this section are all work in progress and the descriptions and examples illustrate ideas rather than final contributions.
Internal Behavior. While $\mathsf{d}\mathcal{L}$ is well-suited to describe hybrid systems, the intent behind variables cannot be specified (e. g., can a controller set a cars acceleration?). We envision a logic based on $\mathsf{d}\mathcal{L}$, extending it by a *type system for variables*. For implementation it is useful to distinguish between *sensors* and *actuators*, *environmental* (laws of physics) and *control* variables (set by choice). For verification, it is useful to distinguish between *readable*, *writeable*, *discrete*, and *continuous* variables. Approaches based on hybrid automata often distinguish between *input*, *control* and *output* variables.

The verification of the internal behavior of a component can make use of variable types, which are usually neglected when proving $\mathsf{d}\mathcal{L}$ formulas. For instance, the controller is only allowed to set certain variables to which it has write access (e. g., set cars acceleration, but not its position). The proof rule[2] in Fig. 1 for assigning the value of a variable y to a variable x in a component c, requires write access to x and read access to y. Similar rules could be derived for other operations (e. g., ODEs), to enforce type safety during proofs.

$$\frac{\Delta, c \downarrow x, c \uparrow y \vdash \phi_x^y, \Gamma}{\Delta, c \downarrow x, c \uparrow y \vdash [x := y]_c\ \phi, \Gamma}$$

Fig. 1. Proof Rule

[2] $c \downarrow x$ means "c has write access to x", $c \uparrow y$ means "c has read access to y"

External Behavior. The external behavior of a component is defined by its *interface*, including *contracts* on input and output ports. Similar to AGR, the interface specifies a contract (e. g., $\psi_1 \rightarrow [C_1]\phi_1$, i. e., assuming ψ_1, all runs of component C_1 guarantee ϕ_1) about what the component assumes at its input ports (assumption ψ_1) and what it guarantees at its output ports (guarantee ψ_1). The contract ψ_1 and ϕ_1 can be specified in various ways. Automata-based AGR approaches (e. g., [5]) mostly use some kind of automaton-based specification. In deductive verification, predicates over the input variables or timed regular expressions are more suitable. Considering the traffic flow example introduced initially, they provide a way of stating which flow is produced by an output for which duration of time and thus allows a detailed interface description (e. g., $(3 \cdot A; 1 \cdot B)*$ means flow A for 3 time units followed by flow B for 1 time unit repeated indefinitely). Note, that every behavior described using timed regular expressions can also be expressed using $\mathsf{d}\mathcal{L}$.

Verification. To ensure formally verified CPS components, two aspects must be considered, namely, verifying that the internal behavior actually follows its specified external behavior (e. g., $\psi_1 \rightarrow [C_1]\phi_1$) and verifying that a component's external behavior is feasible for the operational area at hand and obeys a given local safety condition (e. g., under weaker assumptions Φ_1, stronger promises Ψ_1 are guaranteed, i. e., $(\Phi_1 \rightarrow \phi_1) \wedge (\psi_1 \rightarrow \Psi_1)$, cf. "dominance" in [1]).

In order to model and ensure safety of an entire CPS, we further need a way of safely combining the aforementioned components. Since time always passes simultaneously throughout a hybrid system, components have to be composed in parallel. We envision composition operations that go beyond loss-less instantaneous value passing and propose *composition operators*, including (i) port forwarding operators (i. e., loss-less and instant connection), (ii) operators that influence the continuous evolution of components (e. g., components evolve during communication delay), (iii) operators that affect the forwarded information (e. g., uncertainty for measurements, noise on electrical signals), and (iv) operators that perform state estimation of non-accessible values (e. g., estimate the acceleration by measuring the change of speed). An example composition operation is the noisy composition: $\underset{noise}{\circ\!\!-\!\!\bullet} \equiv \tilde{x} := *; ?\,|\tilde{x} - x| \leq \delta$.

The ultimate goal is to ensure that the composite system obeys to a global safety condition Φ if it is started in a safe state Ψ, i. e., $\Psi \rightarrow [C_1 \circ\!\!-\!\!\bullet C_2]\,\Phi$, which can be achieved by an assume-guarantee style rule for deductive verification using $\mathsf{d}\mathcal{L}$. If it is ensured that a component (e. g., C_1) follows its external behavior specification (i. e., it obeys its contract, e. g., $\psi_1 \rightarrow [C_1]\phi_1$) and ensures its local safety condition (e. g., $\phi_1 \rightarrow \Phi_1$), it remains to ensure provably correct composition, and (if required) automatically derive composition *proof obligations* Θ to ensure overall system correctness (e. g., $(\bigwedge_i \Phi_i) \wedge \Theta \rightarrow \Phi$).

In our initial case study on traffic networks, a composition operation connects two roads allowing cars to drive from one component to another (e. g., from a traffic light to an intersection). For a sample component (e. g., a traffic light) the contract restricts the maximum outflow and the load (i. e., ratio between used and available space on the road) of a component. Here, the local safety

property and the composition property are the same for all components, stating that it should not produce a traffic breakdown for a predefined time (i. e., the number of cars should never exceed the available space) and respectively that the outflow of a component must not be larger than the allowed inflow of the subsequently connected one. If these properties hold, the overall safety property (i. e., no overflow anywhere in the network for a predefined time) can be inferred.

2.3 Evaluation

We plan to implement a software prototype, which will include a library of components and associated composition operations. To show the applicability of the approach, we already implemented a tool called SAFE-T[3]. Based on maximum-flow components it allows combining them to larger traffic networks, while checking the aforementioned arithmetic composition condition automatically. The tool can be used to find the origin of a traffic breakdown and analyze how it will propagate through the network.

Based on the implementation, we will consider existing case studies and compare our compositional models to the original, monolithic ones specifically w.r.t. *model complexity* and *proof effort*.

Acknowledgements. Work funded by BMVIT grant FFG BRIDGE 838526, by OeAD Marietta Blau grant ICM-2014-08600 and as part of P28187-N31.

References

1. Benvenuti, L., Bresolin, D., Collins, P., Ferrari, A., Geretti, L., Villa, T.: Assume–guarantee verification of nonlinear hybrid systems with Ariadne. Int. J. of Robust and Nonlinear Control 24(4), 699–724 (2014)
2. Bogomolov, S., Frehse, G., Greitschus, M., Grosu, R., Pasareanu, C., Podelski, A., Strump, T.: Assume-guarantee abstraction refinement meets hybrid systems. In: Yahav, E. (ed.) Hardware and Software: Verification and Testing, LNCS, vol. 8855, pp. 116–131. Springer (2014)
3. Damm, W., Dierks, H., Oehlerking, J., Pnueli, A.: Towards component based design of hybrid systems: Safety and stability. In: Manna, Z., Peled, D. (eds.) Time for Verification, LNCS, vol. 6200, pp. 96–143. Springer (2010)
4. Davenport, J.H., Heintz, J.: Real Quantifier Elimination is Doubly Exponential. J. Symb. Comput. 5(1-2), 29–35 (1988)
5. Frehse, G., Zhi Han, Krogh, B.: Assume-guarantee reasoning for hybrid I/O-automata by over-approximation of continuous interaction. In: 43rd IEEE Conf. on Decision and Control, CDC. vol. 1, pp. 479–484 Vol.1 (2004)
6. Mitsch, S., Platzer, A.: ModelPlex: Verified Runtime Validation of Verified Cyber-Physical System Models. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification - 5th Int. Conf., RV 2014. LNCS, vol. 8734, pp. 199–214. Springer (2014)
7. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. 20(1), 309–352 (2010)

[3] SAfe Flow-component Editor for Traffic networks,
available online: `http://www.tk.jku.at/people/mueller/publications/itsc15/`

A Novel and Faithful Semantics for Feature Modeling

Aliakbar Safilian

Department of Computing and Software, McMaster University
safiliaa@mcmaster.ca

Abstract. The most common approach to model product lines is feature modeling. Feature models are grouped into basic and cardinality-based feature models. The common understanding of the semantics of feature models is a Boolean semantics. We argue that this semantics does not capture all practically useful information of feature models. To overcome this deficiency, we propose a Kripke semantics for basic feature models and hence the appropriate logic would be a modal logic. As for cardinality-based feature models, we use formal language theory to get a faithful semantics. These ways, we could apply off-the-shelf model checking and formal languages tools, respectively, to do automated analysis over feature models.

1 Research Problems and Objectives

A *product line* is a set of products that share some common features. A feature is "a distinguishable characteristic of a concept that is relevant to some stakeholders" [9]. Product line engineering has many advantages in software design, including a significant reduction in cost and development time [5]. Feature modeling is the most common approach for modeling commonalities and variabilities of product lines. A *feature model* (FM) is a tree of features presenting their *hierarchical* decomposition, called *feature diagram* (FD), with some possible *crosscutting constraints* (CCs) between them. Feature modeling languages are grouped into *basic* and *cardinality-based* FMs. Fig. 1(a) is a basic FM (elec, mnl, atm stand for electric, manual, automated, respect.): edges with filled and unfilled circles denote *mandatory* and *optional* features; filled and unfilled angles denote *OR* and *XOR* groups. In cardinality-based FMs, *cardinalities* on features and groups are used in place of traditional annotations. Fig. 1(b) provides an example of a cardinality-based FM for a grant application system. Cardinality-based FMs are much more expressive than basic ones, since they also specify extra requirements regarding the number of feature instances.

The common understanding of the semantics of an FM in the literature is its product line (a Boolean semantics). However, this semantics loses some essential information of FMs. For a very simple example, consider two FMs $\mathbb{M}_1$ (a is the root and b is the only mandatory child of a) and $\mathbb{M}_2$ (b is the root and a is the only mandatory child of b). $\mathbb{M}_1$ and $\mathbb{M}_2$ represent the same product line consisting of only the product {a, b}, but their hierarchical structures are different.

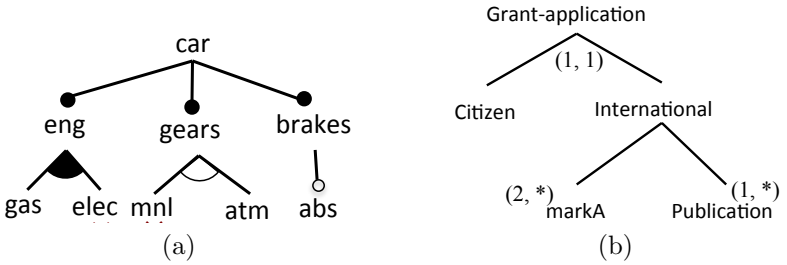

Fig. 1: (a) a basic FM, (b) a cardinality-based FM

The forgotten semantics is important in analysis operations over and reverser engineering of FMs. Indeed, any analysis operation relying on the hierarchical structure of a given FM cannot be addressed using its Boolean semantics. Such analysis operations, including *Least Common Ancestor* of a given set of features, *Sub-features* of a given feature were explicitly characterized in the literature as necessarily relying on this information [2]. Also, the main reason making the current state of the art approach for reverse engineering of FMs [8] heuristic is mainly caused by using such a poor semantics.

Industrial FMs may be very complex involving thousands of features. Hence, they should be represented as formal objects processable by tools. The most common methods to do automated analysis on FMs are based on propositional logic and constraint programming. In these methods, a given FM is translated into propositional logic formulas or constraint programming codes, and then some off-the-shelf tools (e.g., SAT solvers) are used for reasoning about the model. However, these approaches have two drawbacks. First, they cannot support cardinality-based FMs, since such FMs cannot be encoded into these languages. Second, some operations cannot be implemented in these methods because propositional logic and constraint programming translations are based on the Boolean semantics of FMs, which is a forgetful semantics.

To fix these problems, we need to address the following research questions:

Q1) *What is a proper semantics for FMs capturing all interesting and useful aspects of models? What language is appropriate for specifying FMs?*

Q2) *Develop a formal framework for defining the analysis operations on FMs and propose/design an automated tool to support all analysis operations on FMs.*

To address the question **Q1**, we propose a Kripke semantics for basic FMs and hence the appropriate logic would be a modal logic. As for cardinality-based FMs, we use formal language theory to get a faithful semantics. Addressing **Q1** underpin the ability to address **Q2**: we use model checking techniques and off-the-shelf language tools for implementing analysis operations over basic and cardinality-based FMs, respectively.

2 Literature Review

In this section, we briefly review the most common approaches for formalizing the semantics of FMs: propositional logic and context-free grammars for basic and cardinality-based FMs, respectively. We refer the reader to [4] and [6] for a more complete review of the literature.

Using Propositional Logic for Basic FMs: The product line of a given basic FD can be translated into a propositional logic formula generated over the set of features [1]. In this sense, any logical formula can be seen as a CC. The propositional logic encoding enables us to use logic-based tools, such as SAT solvers and Binary-Decision Diagram libraries, for the analysis of FMs. As argued in the first section, the Boolean semantics does not capture all essential information of FMs, which implies that this encoding cannot address all analysis questions over FMs, especially those involving the hierarchical structure of FMs.

Using Context-free Grammars for Cardinality-based FMs: Czarnecki et al, in [3], formalize the semantics of cardinality-based FDs using context-free grammars. The grammar generated for a given cardinality-based FD represents its product line. However, a problem of this procedure is that it gives a left-to-right ordering on siblings (the nodes with the same parent). Such an ordering entails that some syntactically equivalent cardinality-based FDs have different semantics. In addition, generative grammars do not capture the hierarchical structure of cardinality-based FDs. Also, the procedure entirely ignores CCs. This is an essential deficiency, since CCs play a central role in feature modeling.

3 Current Stage of Research

As mentioned in Sect. 1, we use modal logic and formal languages for capturing the semantics of basic FMs and cardinality-based FMs, respectively. Due to the page limitation, we just briefly discuss the modal logic approach. We refer the reader to [7] for the formal languages approach for cardinality-based FMs.

Our observation is that an FM defines not just a set of valid products, but the very way in which these products are to be decomposed step by step from constituent features. Correspondingly, we propose a transition system for FMs, called *partial product lines* (PPLs), initialized at the root feature and gradually progressing towards full products. Fig. 2(c) without the grey elements represents a fragment of the PPL of the FM in Fig. 1(a); the number of letters in the acronym for a feature corresponds to its level in the tree, e.g., c stands for car, en for eng etc. Nodes in a PPL denote *partial* products, i.e., full products with, perhaps, some features missing. The set of full products is a subset of the set of partial products. The PPL of a given FM must satisfy the following requirements:

(i) (tree structure): If a feature is included in a product, then its parent must also be included. For an example, the set {en} is not a valid partial product for the PPL in Fig. 2(a).

(ii) (exclusive constraints): Partial products must satisfy the exclusive constraints. For an example, a set including both atm and mnl is not a valid product.

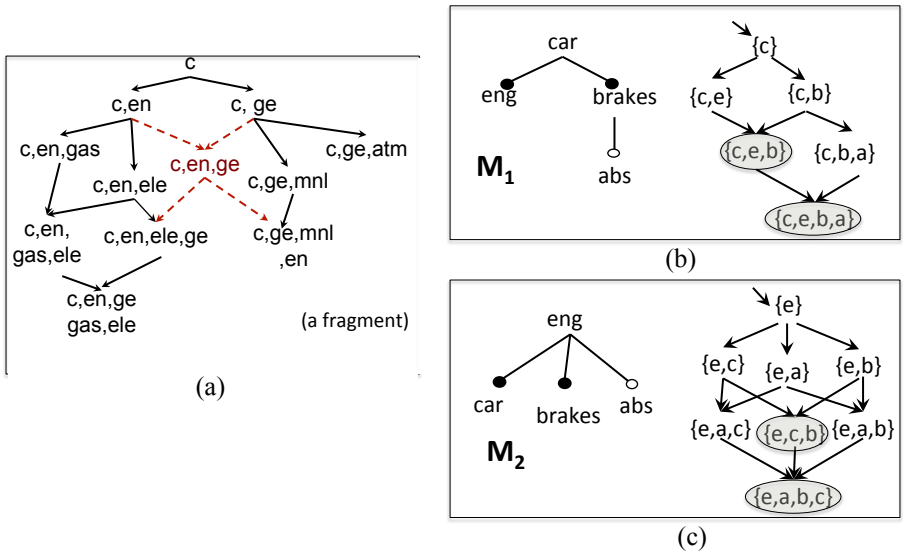

Fig. 2: From BFMs to PPLs

(iii) (Singletonicity): For any transition $P \longrightarrow P'$: $P' \setminus P$ is a singleton set.

(iv) (I2C principle): Processing a new branch of the feature tree should only begin after processing of the current branch has reached a full product. We call this requirement *instantiate-to-completion* (I2C). Importantly, I2C can prohibit some transitions. Obviously, removing some transitions may result in making some products unreachable from the initial product. For an example, consider the partial product $P = \{\mathsf{c}, \mathsf{en}, \mathsf{ge}\}$ in Fig. 2(a). Transition $\{\mathsf{c}, \mathsf{en}\} \longrightarrow P$ is not a valid one, since gear was added to engine *before* the latter is fully assembled. Similarly, transition $\{\mathsf{c}, \mathsf{ge}\} \longrightarrow P$ is also not valid as engine is added before gear instantiation is completed. Hence, P becomes unreachable, and should be removed from the PPL. (in the diagram, invalid edges are dashed.)

Other constraints in the FM influence only full products. We have proven that the above conditions make PPLs faithful semantics counterparts of FMs (please see [4] for the proofs). For an example, consider the FMs and their corresponding PPLs in Fig. 2(b) and (c). The full products are identified by circles in the figure. We see that although both FMs have the same produt line, their PPLs are essentially different, which reflects the essential difference between the FMs.

We distinguish full products by supplying them (the nodes corresponding to), and only them, with identity loops and come to special standard Kripke structures, called *feature Kripke structures* (fKSs). We define *feature computation tree logic* (fCTL), which is a fragment of the computation tree logic (CTL) enriched with a constant modality (for specifying full products) to capture the theory of fKSs.

Given an FM $\mathbb{M}$, we build two fCTL theories, $\Phi_{\mathrm{ML}\subseteq}(\mathbb{M})$ and $\Phi_{\mathrm{ML}}(\mathbb{M})$, such that the former theory is a subset of the latter, and the following statements hold for any fKS K:

Theorem 1 (Soundness). $PPL(\mathbb{M}) \models \Phi_{\mathrm{ML}}(\mathbb{M})$.

Theorem 2 (Semi-completeness). $K \models \Phi_{\mathrm{ML}\subseteq}(\mathbb{M})$ *implies* $K \sqsubseteq PPL(\mathbb{M})$.

Theorem 3 (Completeness). $K \models \Phi_{\mathrm{ML}}(\mathbb{M})$ *iff* $K = PPL(\mathbb{M})$.

In Theorem 2, $\sqsubseteq$ denotes the substructure relation, i.e., $K \sqsubseteq PPL(\mathbb{M})$ means that K is a substructure of $PPL(\mathbb{M})$. Completeness allows us to replace FMs by the respective fCTL-theories, which are highly amenable to formal analysis and automated processing. Semi-completeness is useful (as an auxiliary intermediate step to completeness, but also) for some important practical problems in FM such as *specialization* [10] ($\mathbb{M}$ is a specialization of another FM $\mathbb{M}'$ if the product line of $\mathbb{M}$ is a subset the product line of $\mathbb{M}'$), and some other analysis operations [2] over FMs. These operations are normally considered for full product lines (FPLs) only, but can be redefined for PPLs as well.

References

1. D. Batory. *Feature models, grammars, and propositional formulas.* Springer, 2005.
2. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.
3. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
4. Z. Diskin, A. Safilian, T. Maibaum, and S. Ben-David. Modeling product lines with kripke structures and modal logic. (GSDLab TR 2014-08-01), 08/2014 2014.
5. K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques.* Springer, 2005.
6. A. Safilian, T. Maibaum, and Z. Diskin. The semantics of feature models via formal languages (extended version). (GSDLab TR 2014-08-02), 08/2014 2014.
7. A. Safilian, T. Maibaum, and Z. Diskin. The semantics of cardinality-based feature models via formal languages. To appear in proceedings of FM 2015: Formal Methods, 2015.
8. S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE 2011*, pages 461–470. IEEE, 2011.
9. M. Simos, R. Creps, C. Klingler, and L. Lavine. Software technology for adaptable reliable systems (stars). organization domain modeling (odm) guidebook, version 1.0. Technical report, DTIC Document, 1995.
10. T. Thum, D. Batory, and C. Kastner. Reasoning about edits to feature models. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 254–264. IEEE, 2009.

A Formal Model for the SCJ Level 2 Paradigm

Matt Luckcuck
Department of Computer Science, University of York, UK
ml881@york.ac.uk

1 Introduction

Safety-Critical Java (SCJ) [12] is the product of an international effort to provide a Java-based language for applications that must be certified using the avionics standard ED-12C/DO-178C [4] at Level A, which defines software that would prevent continuous safe flight and landing in the event of failure. To aid certification, SCJ is organised into three compliance levels that ascend in complexity from Level 0 to Level 2.

The SCJ standard does not cover verification techniques. Verification has been addressed and results obtained for Level 1, but not Level 2. We focus on providing verification for SCJ Level 2 programs. SCJ Level 2 has received little attention from practitioners and researchers, even its intended uses are unclear from the standard, and in [14] we present the first examination of the uses of its features and present example applications for Level 2.

The SCJ API ensures a hierarchical program structure and supports several real-time execution abstractions. SCJ programs are centred around missions, which each contain several real-time tasks that perform a particular function. Uniquely for SCJ, a Level 2 program may have many concurrent missions, which allows Level 2 programs to adopt more complex structures than those at the other two compliance levels. Tasks from any active mission may preempt each other, based on their priorities; there is no assumption that tasks from a particular mission have precedence. Level 2 tasks may use all four SCJ execution patterns: periodic, aperiodic, run-once after a time offset, and run-to-completion. Finally, Level 2 programs may use the familiar Java suspension features.

Our work makes three contributions to the state of the art on verification of SCJ Level 2 programs. Firstly, we model the SCJ Level 2 paradigm using the state-rich process algebra *Circus* [15]. Our model can be used to identify potential errors in the programs that it represents. *Circus* combines Z [10] for modelling state, CSP [7] for modelling behaviour, and Morgan's refinement calculus [9]. A *Circus* program is organised around processes, which may have a state component to hold variables and actions to perform behaviours. Communication between processes is achieved via CSP channels. Our model uses features from other members of the *Circus* family. *OhCircus* [2] introduces a notion of object orientation and inheritance, and we use features from *Circus* Time [13] to specify time budgets and deadlines.

We provide a mechanised translation strategy that enables the automatic transformation of SCJ Level 2 source code into faithful *Circus* models. As a secondary objective, we also provide a strategy for translating our models back into SCJ programs.

Our second contribution rests on our model capturing the API separately from the program-specific behaviour. Because of this separation we can show that the SCJ API does not introduce undesirable behaviour, such as deadlock or livelock, under the circumstances that we capture.

There is a body of previous work involving *Circus* and SCJ, including a model of SCJ Level 1 [16] – upon which our work is based. A refinement strategy [3] has been devised to transform abstract specifications into concrete specifications that capture the SCJ paradigm. This refinement strategy facilitates the development of SCJ programs that are correct by construction.

Our final contribution is that our model provides the refinement strategy [3] with a target for models of SCJ Level 2. While this refinement strategy is out of scope for our work, our model enables it to consider Level 2 programs.

Previous approaches to ensuring the safety of SCJ programs include using annotations to provide run-time checks [11] or to specify checkable program constraints [6]. RSJ [8] is a tool that explores all possible schedulings of the threads within an SCJ program to check for scheduling-dependent errors. However, none of these techniques are specifically aimed at Level 2.

ABS [1] is an executable specification language that has similar capabilities to *Circus*. Both ABS and *Circus* have an object-oriented model that is similar to Java's and capture concurrency. However, *Circus* contains a notion of refinement that ABS does not. Refinement is important for our third contribution.

In the next section we describe our model of SCJ and what analysis it facilities. Finally, in Section 3 we summarise our research and contributions, and describe the further work needed to complete this research.

2 Model and Translation

We capture the paradigm of SCJ Level 2, agnostically of its implementation in Java, using two components. The framework model captures the behaviour of the API classes of SCJ and is reused for each program. Conversely, each program is represented by an application model that captures its particular behaviour.

The framework and application models both contain a process for each of the SCJ API classes. The framework processes control the program flow and hand off to their application counterparts wherever the program runs application code, including where API methods are overridden.

We capture Java exceptions but only when they indicate a misuse of the SCJ paradigm, never when they indicate a purely Java problem (such as a `null` parameter). If the program uses locking or suspension, then we capture this in extra elements added to the framework model.

The translation strategy that we are developing contains formal rules that build the specification of a given program. Our work provides the first formal semantics of SCJ Level 2. As there is nothing else formal to compare our semantics to, we can not consider its soundness, but it will be validated using tools and case studies.

A *Circus* model checker is in development; in the mean time we translate our *Circus* model into CSP to validate our specifications using FDR3 [5]. We animate its behaviour and compare it to that described in the SCJ standard. We model check it to identify properties (such as deadlock, livelock, and non-termination) that represent program errors and SCJ-specific problems, such as Java exceptions that indicate a misuse of the SCJ paradigm. This also gives us confidence that our model of the SCJ infrastructure is correct and helps to verify the SCJ API itself, because we model it separately.

Our approach is limited to capturing the behaviour of SCJ programs. We do not capture use of resources, in particular memory usage. Further, while we capture time for the purposes of deadline detection, our models cannot be used to calculate the worst-case execution time of a program.

We have used FDR3 to show that our model of the SCJ infrastructure is free from program errors, meaning that if our specification of a program exhibits these errors, then they must arise from the application model. Further, we have translated several small example applications into our model, by hand, to show that our model can capture the SCJ Level 2 paradigm. Using FDR3 we have proved that these examples do no throw exceptions, are free from deadlock and livelock, and that they terminate.

3 Summary and Further Work

In summary, we model the paradigm of SCJ Level 2 as a combination of a framework model, that captures the SCJ API, and an application model, that captures the behaviour of the program being modelled. Our model of SCJ Level 2 contributes to both top-down development of correct SCJ programs, as a target for the refinement strategy presented in [3], and to bottom-up development of correct SCJ programs, as a tool for the identification of program errors. Further, it can be used to verify the SCJ API because we capture it separately in our model.

Our framework model and the skeleton processes for the application model are both complete. We have modelled several small example programs, to show that we can capture the common features of the SCJ Level 2 paradigm, and shown that these examples do not exhibit any undesirable properties.

The remaining work is to formalise the translation of a program into our model. Translation will then be automated using a tool that will take SCJ programs as an input and output *Circus* models. We envisage minor restrictions on the form of the SCJ programs, similar to those presented in [16]; for example, each SCJ class should be in its own file. Automatic translation not only validates our model, but also enables the verification of SCJ Level 2 programs, by allowing a simple translation to our model to enable model checking. More work on analysing our model is needed and the basic properties we can already prove will be augmented by properties that capture SCJ exceptions that indicate a misuse of the paradigm and application-specific properties.

Acknowledgements

This work is funded by the hiJaC project, backed by the EPSRC grant EP/H017461/1. We would like to thank Ana Cavalcanti, Andy Wellings, Frank Zeyda, Alan Burns, and Thomas Gibson-Robinson.

References

1. Bubel, R., Montoya, A.F., Hähnle, R.: Analysis of executable software models. In: Formal Methods for Executable Software Models, pp. 1–25. Springer (2014)
2. Cavalcanti, A., Sampaio, A., Woodcock, J.: Unifying classes and processes. Software & Systems Modeling 4(3), 277–296 (2005)
3. Cavalcanti, A., Wellings, A., Woodcock, J., Wei, K., Zeyda, F.: Safety-critical Java in Circus. In: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 20–29. JTRES '11, ACM, New York, NY, USA (2011)
4. EUROCAE and RTCA: Software Considerations in Airborne Systems and Equipment Certification. Norm ED-12C, EUROCAE (2012)
5. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: Failures Divergences Refinement (FDR) Version 3 (2013)
6. Haddad, G., Hussain, F., Leavens, G.T.: The design of SafeJML, a specification language for SCJ with support for WCET specification. In: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 155–163. JTRES '10, ACM, New York, NY, USA (2010)
7. Hoare, C.A.R.: Communicating Sequential Processes.
8. Kalibera, T., Parizek, P., Malohlava, M., Schoeberl, M.: Exhaustive Testing of Safety-Critical Java. In: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 164–174. JTRES '10, ACM, New York, NY, USA (2010)
9. Morgan, C.: Programming from specifications. Prentice-Hall, Inc. (1990)
10. Spivey, J.M.: The Z Notation: A Reference Manual. International Series in Computer Science (1992)
11. Tang, D., Plsek, A., Vitek, J.: Static checking of safety critical java annotations. In: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 148–154. ACM, Prague, Czech Republic (2010)
12. The Open Group: Safety-Critical Java Technology Specification. Tech. rep., The Open Group (27 December 2014)
13. Wei, K., Woodcock, J., Cavalcanti, A.: New circus time. University of York, Tech. Rep., February (2012)
14. Wellings, A., Luckcuck, M., Cavalcanti, A.: Safety-Critical Java Level 2: Motivations, Example Applications and Issues. In: Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems. pp. 48–57. JTRES '13, ACM, New York, NY, USA (2013)
15. Woodcock, J., Cavalcanti, A.: The Semantics of Circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002:Formal Specification and Development in Z and B, Lecture Notes in Computer Science, vol. 2272, pp. 184–203. Springer Berlin Heidelberg (2002)
16. Zeyda, F., Lalkhumsanga, L., Cavalcanti, A., Wellings, A.: Circus Models for Safety-Critical Java Programs. The Computer Journal (2013)

A Code Generator for VDM-RT models

Miran Hasanagić

Department of Engineering, Aarhus University,
Finlandsgade 22, 8200 Aarhus N, Denmark
miran.hasanagic@eng.au.dk

1 Introduction

The research presented in this abstract is part of my PhD: "Tool Automation for Model Based Design of Cyber Physical Systems". This research is based on the formal method Vienna Development Method (VDM) [6, 2], which supports specification, modelling and analysis of software systems in a discrete time domain. Currently three dialects exist, VDM-SL, VDM++ and VDM-RT. VDM-SL enables modelling of functional specification of sequential systems, while VDM++ extends it, and supports object-oriented modelling by introducing classes [3]. Finally, VDM-RT extends VDM++, and supports the modelling of *distributed* real time systems [15].

This research is based on the VDM-RT notation, which has shown to be advantageous when developing Distributed Systems (DSs) [15, 14]. More specifically, this research focuses on the process of implementing a DS modelled in VDM-RT into a programming language. The main goal of this work is to automate the implementation process of a VDM-RT model by developing a Code Generator (CG). This has not been researched before, and in the current approaches a VDM-RT model is implemented manually [10].

The main motivation for generating code automatically is that it can reduce the development time, even though it may be required to adjust it manually after the code has been generated. Also a manual implementation may introduce inconsistency between the formal model and the implemented code. Such a CG may enable the designer to spend more time in the modelling phase during development, hence increase reliability without increase in the development time.

2 Problem being addressed

As indicated in the introduction, the problem being addressed is how to automate the process of implementing a VDM-RT model to a programming language. The CG discussed here, uses the distribution technology Java Remote Method Invocation (RMI) [13] in order to implement the distributed aspects of a VDM-RT model. This CG generates code that supports communication between distributed entities which generate network communication in a VDM-RT model. Both VDM-RT and Java RMI are presented briefly below.

As a consequence of the lossness in a VDM-RT model, multiple valid interpretations of the model may exists. However, the interpretation of a VDM-RT model is

on purpose deterministic even when modelling concurrency aspects [11]. Due to the modelling lossness and that the execution of Java code is non-deterministic, a CG may generate Java code which may not follow the same execution path for each simulation of the same test case. However, each execution of the generated Java code shall be one of the possible interpretations of a VDM-RT model.

3 Distributed Systems in VDM-RT

VDM-RT enables the designer to model a DS as a single system during the modelling phase. As stated above, the VDM-RT notation is an extension to the object-oriented VDM++ notation. Hence objects are the communicating entities when modelling a DS. Beside having classes, VDM-RT adds the notion of a system definition. In the system definition computational elements can be created, called CPUs, and communications channels between these CPUs, called BUSses. So inside this system definition the distributed aspects of VDM-RT are introduced.

A DS is modelled by instantiating CPUs, deploying object to each CPU in order to model its functionality, and finally modelling the system architecture by connection CPUs using BUSes in order to enable communication. The objects instantiated inside the system defintion are globally accessible, and each VDM-RT model only has one system definition. Hence these objects can be used anywhere inside the model. However, in order for two objects deployed on two different CPUs to communicate, the CPUs have to be connected by a BUS. It shall be noted that VDM-RT also introduces the notion of time, but this has currently not been addressed by this research.

The interpretation of a VDM-RT model is initiated from a special virtual CPU, that is connected to all other CPUs inside the system defintion. This virtual CPU is meant only be used for modelling the expected environment and storing test results [10].

4 Java RMI

Java RMI enables objects instantiated on different Java Virtual Machines (JVMs) to communicate transparently. In order for an object to be remotely accessible it has to be instantiated from a Java class which implements a Java interface that enables Java RMI communication. This is achieved by having both the class and its corresponding interface extending Java RMI relevant properties as defined by [13]. The methods defined in the interface become remotely accessible. References to remote objects can be obtained by using a registration service (server), in which objects can be stored and lookup by a unique name.

5 Research Outcome

The research outcome is to enable transparent communication between objects in a VDM-RT model when generated to code by using Java RMI. Additionally, a research outcome is to identify general solutions, by developing this CG, that can be reused for other CGs using another technology for enabling network communication.

Since both VDM-RT and Java RMI build upon the RMI communication paradigm, object communicating transparently, Java RMI is a good first choice. Additionally, Java RMI is chosen in order to use the existing VDM++-to-Java CG [7], which is part of the Overture platform [9], for generating code for the functionality of a single CPU. This is possible since the collection of objects deployed to a single CPU can be viewed as a single VDM++ model, which possibility is depended on objects located on another CPU. Hence the existing VDM++-to-Java CG can generate the functionality of each CPU, while the CG presented here enables the network communication between CPUs.

The main objective of this CG is to preserve the semantics in a VDM-RT model when generating code in order to support network communication. This CG implements each created CPU in a VDM-RT model as an individual JVM. Every realised CPU gets its own local system class, which only contains its local and remote objects according to the VDM-RT model. This solves the problem of how to cope with the challenge that a VDM-RT model has a single system definition for the whole DS model.

In a VDM-RT model both remote and local objects of a single CPU are the actual class type. So a difference between a VDM-RT model and Java RMI code, is that a local object uses the actual class implementation, while a remote object is referenced using the interface. Hence the CG has to generate a Java class that contains all the functionality of a VDM-RT class, and a corresponding Java RMI interface which only contains the signatures of the public methods of the class. This enables the CG to represent both an object locally by the actual class, and the object remotely by its interface definition. However, because VDM-RT does not distinguish between types of local and remote objects, the CG is required to transforms a local class with the equivalent remote interface in order to support both local and remote objects to passed as method arguments in the generated code.

Finally, before the main execution of each CPU can be started, all objects are required to be instantiated on the correct CPUs, and each CPU has to obtain references to its remote objects. Hence the CG is required to generate an initialisation mechanism such as the VDM-RT interpreter has before evaluating the model. Currently, in order to initialise all objects a registration service that is connected to all CPUs is used, where all object inside the system defintion in a VDM-RT model are stored and looked up. Additionally, since the interpreter starts the evaluation from the virtual CPU, a limitation is to ensure that the VDM-RT model only is started by objects deployed to real CPUs. These objects can then be placed inside the entry function of the CPU where they are deployed in the generated code. More in-detail and technical information about how the Overture[1] VDM++-to-Java CG is extended is described in [5].

6 Expected Contributions

Generating code for a VDM-RT model is a novel area of research, hence the research of using Java RMI as a distributed technology is a first step of research towards code generating all aspects of a VDM-RT model. Java can not be used to support real time on a single CPU, and Java RMI can itself not be used for real time communication.

[1] www.overturetool.org

However, it provides some solutions of implementing the distributed aspects of a VDM-RT model that can be generalised in order to be reused when enabling distribution by other technologies using RMI, such as CORBA [12].

This research helps additionally understanding the limitations of a VDM-RT model when it comes to an actual implementation of a modelled DS. It can support to propose extensions and identify a subset of the VDM-RT language that enables full code generation.

7 Related Research

Generating code for the distributed aspects of VDM-RT has not been researched before, so currently there exists no related work to the problem being addressed in the research presented here. There exist, however, CGs for VDM++ as described in [7, 8].

Code generation in order to support real time tasks is discussed in [1], which can be relevant when the real time aspects of a VDM-RT model have to be supported. Additionally, [4] presents how code can be generated for Event-B models. However, for the research presented in this abstract the focus mainly has been to illustrated that the distributed aspects can be supported by a code generator, while preserving the semantics of VDM-RT.

8 Current Progress and Publications

The CG presented here has been validated by a case study. Currently this CG does not support of all aspects of a VDM-RT model, such as time, periodic threads, CPU speed and BUS bandwith. For this reason, the current CG is limited to preserving operation ordering within a CPU. This CG is, however, used in order to show that a VDM-RT model can be code generated. Future research will address how to incorporate timing aspects and use other distribution technologies in combination with other programming languages inside the EU INTO-CPS project[2].

Currently the only relevant publication of this author for this research area is [5], which is submitted for the Overture Workshop at FM15. The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047.

References

1. Amnell, T., Fersman, E., Petterson, P., Sun, H., Yi, W.: Code synthesis for timed automata. Nordic Journal of Computing (2003)
2. Bjørner, D.: Pinnacles of software engineering: 25 years of formal methods. Annals of Software Engineering 10, 11–66 (2000)
3. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), http://overturetool.org/publications/books/vdoos/

[2] www.into-cps.au.dk

4. Furst, A., Hoang, T.S., Basin, D., Desai, K., Sato, N., Miyazki, K.: Code generation for event-b
5. Hasanagić, M., Larsen, P.G., Jørgensen, P.W.V.: Generating java rmi code for the distributed aspects of vdm-rt models. Submitted to the Overture Workshop at FM15 (2015)
6. Jones, C.B.: Scientific Decisions which Characterize VDM. In: Wing, J., Woodcock, J., Davies, J. (eds.) FM'99 - Formal Methods. pp. 28–47. Springer-Verlag (1999), lecture Notes in Computer Science 1708
7. Jørgensen, P.W., Couto, L.D., Larsen, M.: A Code Generation Platform for VDM. In: The Overture 2014 workshop (June 2014)
8. Larsen, P.G.: Ten Years of Historical Development: "Bootstrapping" VDMTools. Journal of Universal Computer Science 7(8), 692–709 (2001)
9. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
10. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Embedded Systems using VDM. Intl. Journal of Software and Informatics 3(2-3) (October 2009)
11. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Qin, S., Qiu, Z. (eds.) Proceedings of the 13th international conference on Formal methods and software engineering. Lecture Notes in Computer Science, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (October 2011), http://dl.acm.org/citation.cfm?id=2075089.2075107, ISBN 978-3-642-24558-9
12. OMG: The Common Object Request Broker: Core Specification. (November 2002)
13. Sun: Java Remote Method Invocation Specification (2000)
14. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2009)
15. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085, Springer-Verlag (2006)

Privacy-Preserving Social Networks

Raúl Pardo

Dept. of Computer Science and Engineering,
Chalmers University of Technology, Sweden.
pardo@chalmers.se

1 Outline of the problem

One of the aims of social networks (SNs) is to be flexible in the way one shares information, being as permissive as possible in how people communicate and disseminate information. While preserving the spirit of SNs, users would like to be sure that their privacy is not compromised. Though limiting the capability of users in what concerns what they can share or not might be in the interest of service providers in general, and in particular of SNs, we believe citizens should be in power to control and decide on how much information to make public. One way to do so is by providing users with means to define their own privacy policies and give guarantees that they will be respected. Privacy in SNs may be compromised in different ways: from direct observation of what is posted (seen by non-allowed agents), by inferring properties of data (*metadata privacy leakages*), indirectly from the topology of the SN (e.g., knowing who our friends are), to more elaborate intentional attackers such as *sniffers* or *harvesters* [6]. Among others, one of the origins of these attacks comes from their privacy enforcement mechanism, the so called Relationship-Based Access Control (ReBAC) [5].

We aim for developing a privacy enforcement mechanism which offers social network users the possibility of expressing finer-grained privacy policies, enabling them to deal with (certain kinds of) implicit disclosure of sensitive information. As a first step towards that goal, we have developed the privacy policy framework $\mathcal{PPF}$ for social networks [7], which is briefly described in next section.

2 The Privacy Policy Framework $\mathcal{PPF}$

The privacy policy framework $\mathcal{PPF}$ for social networks [7] consists of:

A social network model. $\mathcal{SN}$ is a *social graph*, a graph whose nodes represent users, and edges represent different kind of relationships between users. The graph is enriched with information on the knowledge the users of the social network have, and what they are permitted to do.

A knowledge-based logic. $\mathcal{KBL}_{\mathcal{SN}}$ is an epistemic-deontic logic which provides the possibility to reason about what the agents know and what they are allowed to do. The logic allows us not only to access and reason about the explicit knowledge of an agent, but also about implicit knowledge (through inferences).

A formal privacy policy language. $\mathcal{PPL}_{\mathcal{SN}}$ is a language for writing privacy policies for each individual user.

Besides, the framework also comes with a *satisfaction relation* defined for the logic $\mathcal{KBL}_{\mathcal{SN}}$, and a *conformance relation* defined for the policy language $\mathcal{PPL}_{\mathcal{SN}}$.

The framework may be tailored by providing suitable instantiations of the different relationships, the events, the propositions representing what is to be known, and the additional facts or rules a particular social network should satisfy. In order to show how $\mathcal{PPF}$ can be used, we have instantiated the privacy policies of Facebook and Twitter [7], which are two of the most used social networks nowadays. For instance, one of Facebook's privacy policies is responsible for setting the audience of a post, where the user can choose among 'Friends', 'Only me' and 'Custom'. In $\mathcal{PPF}_{\text{Facebook}}$ it would be split in 3 policies. In the mentioned instantiation if u wants the audience of her posts to be her Friends, it would be written as follows:

$$[\![\neg S_{Ag \setminus friends(u) \setminus \{u\}} u.post_n^j]\!]_u$$

where $S_G\phi$ is a formula stating "somebody in the group G knows ϕ", Ag is the set of all the agents in $\mathcal{SN}$, $u, j \in Ag$, $n \in \mathbb{N}$, $post_n^j$ represents post n in j's timeline and $friends(u)$ is an function which returns all the friends of u.

In $\mathcal{PPF}$ it is possible to write more expressive policies: we can choose any attribute we like and define an audience for it. For example, we can write the following privacy policy:

Only my friends can know the posts I liked

which would be written in $\mathcal{PPL}_{\mathcal{SN}}$ as follows

$$[\![\neg S_{Ag \setminus friends(u) \setminus \{u\}} u.like_{post_n^j}]\!]_u.$$

As we mentioned before $\mathcal{PPF}$ is an generic framework, therefore we could combine instantiations of two (or more) different social networks in one. This is a very useful and innovative feature, since currently it is becoming more common to connect several accounts from different social networks and share information between them. As a final example of the use of our framework, we present below an example of a privacy policy concerning the combination of $\mathcal{PPF}_{\text{Twitter}}$ and $\mathcal{PPF}_{\text{Facebook}}$ [7]. The following privacy policy:

Only my friends in Facebook who are following me in Twitter can know my location

will be written in our formalism as

$$[\![\neg S_{Ag \setminus (friends(u) \cap Followers(u)) \setminus \{u\}}\ u.location]\!]_u.$$

In $\mathcal{PPF}$, we do not follow the traditional semantics for epistemic logic [4]. It is because the traditional approach is based on modelling the uncertainty

of the agents. If one try to model the uncertainty of the millions of users of a social network, it would require a gigantic state space [9], [8]. Instead we chose to explicitly model what the agents know, which considerably reduces the state space required to model the users' knowledge [7] and leads to a more practical approach.

As we mentioned in Section 1, SNs implement the access control model ReBAC. Fong *et al.* introduced a formalism, which aimed at providing a better understanding of ReBAC [5]. In ReBAC users define an audience to their resources based on relationships, e.g. "My posts can be accessed only by my friends". However, it does not appropriately protect against implicit disclosure of information. For instance, imagine that Alice defines the policy "Nobody can know my location". If now Bob posts the message "I'm in Sweden with Alice", Alice policy would not be enforced, since the message is a resource of Bob. In $\mathcal{PPF}$, Alice's location would be protected independently of who is disclosing the information. The main advantage of ReBAC is its efficiency, since it only requires to check whether the user trying to access the resource is part of the audience. Fong *et al.* also defined a language based on Hybrid logic to express privacy policy in ReBAC [3]. A comparison between the expressiveness of this language and $\mathcal{PPL}_{\mathcal{SN}}$ is part of our future work.

3 Current and Future Work

$\mathcal{PPF}$ offers a framework in which, given a "static" $\mathcal{SN}$ we can check if a set of privacy policies is conformance with it. However social networks evolve as their users execute events. A social network user can make new friends, post new pictures, share her location, etc.

Currently we are working on the dynamic version of $\mathcal{PPF}$, $\mathcal{PPF}^{\mathcal{D}}$. In the dynamic framework we add to $\mathcal{PPF}$ the definition of a set of inference rules which determine how the knowledge and the permissions of the agents evolve depending on the events they execute. We define 4 types of rules:

- *Knowledge-based.* These rules describe how the knowledge and the permission change when executing the rule.
- *Social topology.* These rules modify the social topology of the $\mathcal{SN}$, i.e. the users and their relationships. For instance, adding new users, relationships between them, etc.
- *Policy.* These rules allow for the modification of the set of privacy policies of the agents.
- *Hybrid.* These are special rules, in which it is allowed to combine elements from the three previous types of rules.

The addition of the dynamic this aspect leads to new and more interesting results. Given a set of inference rules of an instantiation and a set of privacy policies, we can formally prove that the set of rules is privacy-preserving with respect to the set of privacy policies (i.e. all privacy policies are preserved under any possible event of the social network). We have written the rules for Twitter

and prove that it preserves privacy with respect to the set of privacy policies defined in [7] for $\mathcal{PPF}_{\text{Twitter}}$. Moreover, we proved that Twitter and Facebook would not handle the addition of some natural privacy policies. Specifically, we proved that adding the policy "It is not permitted that I am mentioned in a tweet which contains a location" would not be protected by the current behaviour of Twitter. Besides, we partially defined the set of rules for Facebook and proved that if we add the policy "I can only be tagged in a picture if I have approved it" it would not be preserved.

Our next step is extending $\mathcal{PPF}^{\mathcal{D}}$ with real time policies. As I mentioned we would like to provide the user with fine-grained control when protecting her information. Real time policies will allow users to specify time frames when the information is available. For example, a privacy concern user would like to enforce the following policy "My boss can never know my location after 20:00".

As mid-term goal, we aim at developing run-time monitoring techniques to enforce that privacy is preserved online, both for evolution concerning the network as well as the policies themselves. Obviously, a centralised version of the monitor would not be practical, so we aim at implementing a distributed monitor. We have already implemented a very limited prototype of $\mathcal{PPF}^{\mathcal{D}}$ in the open source social network *Diaspora** [1][2]. However, we planning to implement the full policy framework and the run-time enforcement mechanism.

References

1. Diaspora*. https://joindiaspora.com/. Accessed: 2015-05-17.
2. $\mathcal{PPF}$ diaspora*. https://github.com/raulpardo/ppf-diaspora. Accessed: 2015-04-12.
3. G. Bruns, P. W. Fong, I. Siahaan, and M. Huth. Relationship-based access control: its expression and enforcement through hybrid logic. In *CODASPY'12*, pages 117–124. ACM, 2012.
4. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about knowledge*, volume 4. MIT press Cambridge, 1995.
5. P. W. Fong. Relationship-based access control: Protection model and policy language. In *CODASPY'11*, pages 191–202. ACM, 2011.
6. B. Greschbach, G. Kreitz, and S. Buchegger. The devil is in the metadata - new privacy challenges in decentralised online social networks. In *PerCom Workshops*, pages 333–339. IEEE, 2012.
7. R. Pardo and G. Schneider. A formal privacy policy framework for social networks. In *SEFM'14*, volume 8702 of *LNCS*, pages 378–392. Springer, 2014.
8. J. Ruan and M. Thielscher. A logic for knowledge flow in social networks. In *AI 2011: Advances in Artificial Intelligence*, pages 511–520. Springer, 2011.
9. J. Seligman, F. Liu, and P. Girard. Facebook and the epistemic logic of friendship. In *TARK'13*, 2013.

Index of Authors