Belgacem BEN HEDIA, CEA, LIST, France
Florin Popentiu VLADICESCU, University of Oradea, Romania

VECoS'2015

Verification and Evaluation of Computer and Communication Systems

9th International Workshop

Bucharest, Romania, September 10-11, 2015

Proceedings

©Copyright 2015 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

These proceedings are published online by the editors as Volume 1431 at
CEUR Workshop Proceedings
ISSN 1613-0073
http://ceur-ws.org/Vol-1431

PREFACE

These are the proceedings of the 9th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'2015) held the 10th and 11th of September 2015 at the University POLITEHNICA of Bucharest that is the main organizer together with the support of MeFoSyLoMa group and Formal Methods Europe.

The International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS) was created by an Euro-Maghrebian network of researchers in computer science. Its first edition, VECoS 2007, took place in Algiers, VECoS 2008 in Leeds, VECoS 2009 in Rabat, VECoS 2010 in Paris, and VECoS 2011 in Tunis, VECoS 2012 in Paris, VECoS 2013 in Florence and VECoS 2014 in Bejaia. The aim of the VECoS workshop is to bring together researchers and practitioners, in the areas of verification, control, performance, quality of service, dependability evaluation and assessment, in order to discuss the state-of-the-art and the challenges in modern computer and communication systems in which functional and extra-functional properties are strongly interrelated. Thus, the main motivation for VECoS is to encourage the cross-fertilization between the various formal verification and evaluation approaches, methods and techniques, and especially those developed for concurrent and distributed hardware/software systems. Beyond its technical and scientific goals, another main purpose of VECoS is to promote collaboration between participants in research and education in the area of computer science and engineering. We welcome contributions describing original research, practical experience reports and tool descriptions/demonstrations in the areas of verification, control, performance, quality of service and dependability evaluation.

The invited speakers for VECoS 2015 are: **Liliana Cucu-Grosjean** from AOSTE, INRIA Paris-Rocquencourt, France, **Gabriel Ciobanu** from Romanian Academy, ICS, Iasi, Romania and **Mohamed Kaâniche** from LAAS, Toulouse, France.

We received 15 high-quality contributions. For each paper three to five reviews were made. The program committee has accepted 9 papers for full presentation.

Without support of our academic and corporate sponsors, the enormous efforts of authors, reviewers, steering committee and the local organizational team this workshop wouldn't provide such an interesting booklet.

We thank the authors for their submissions and the program committee for their hard work.

September 2015 Belgacem Ben Hedia and Florin Popentiu Vladicescu

ORGANIZING COMMITTEE

Professor Nicolae Tăpus, Computer Science Department, The Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Bucharest, Romania
Professor Viorel-Puiu Paun, Faculty of Applied Sciences, University POLITEHNICA of Bucharest, Bucharest, Romania

STEERING COMMITTEE

Hassane Alla , GIPSA Lab INPG Grenoble
Djamil Aissani, LAMOS, Université de Bejaia
Kamel Barkaoui , CEDRIC CNAM Paris (**Chair**)
Hanifa Boucheneb, Veriform, Ecole Polytechnique de Montréal
Francesco Flammini, Ansaldo STS, Milano
Mohamed Kaaniche, LAAS CNRS, Toulouse
Bruno Monsuez , ENSTA - UIIS, Paris
Nihal Pekergin , LACL UPEC, Créteil
Denis Poitrenaud, LIP6 UPMC, Paris
Tayssir Touili, LIAFA, Université Paris Diderot

PROGRAM COMMITTEE

Bernhard K. Aichernig, ISF, Graz
Djamil Aissani, LAMOS, Bejaia
Yamine Ait Ameur, IRIT/ENSEEIHT, Toulouse
Otmane Ait Mohamed, Concordia University, Montréal
Hassane Alla, GIPSA, Grenoble
Lamia Atma Djoudi, Synchrone Technologies, Paris
Kamel Barkaoui, Cedric Cnam, Paris
Zohra Bakkoury, EMI EAMPIS, Rabat
Belgacem Ben Hedia, LIST-CEA, Saclay (**co-chair**)
Saddek Bensalem, VERIMAG Grenoble, France
Simon Bliudze, EPFL, Lausanne
Jean-Louis Boimond, LARIS, Angers
Patrice Bonhomme, LI, Tours
Abdelmadjid Bouabdallah, Heudiasyc Lab, Compiegne
Hanifa Boucheneb, Polytechnique Montréal
Florian Brandner, ENSTA ParisTech, Saclay
Carla Ceatzu, University of Cagliari
Tijani Chahed, Telecom SudParis, Evry
Feng Chu, IBISC, Evry
Isabel Demongodin, LSIS Marseille
Josée Desharnais, Université Laval, Quebec
Karim Djouani, LISSI UPEC, Créteil
Mohamed Escheikh, ENIT,Tunis
Alessandro Fantecchi, Unifi, Italy
Francesco Flammini, Ansaldo STS, Milano
Mohamed Ghazel, INRETS, Villeneuve d'Ascq
Latéfa Ghomri, University of Tlemcen.
Bernd Heidergott, VU Amsterdam University
Serge Haddad, LSV Cachan
Awatef Hicheur Cairns, Altran Research, Velizy
Malika Ioulalen, USTHB Alger
Mohamed Jmaïel, ENSI, Sfax
Mohamad Jaber, AUB, Beyrouth
Jorge Julvez, University of Zaragoza
Mohamed Kaaniche, LAAS, Toulouse
Lars Michael Kristensen, Bergen Univ. College, Bergen
Bechir Ktari, Université Laval, Québec
Gaiyun Liu, Xidian Univ., Xi'an
Borhen Marzougui, ECT, Abu Dhabi
Mourad Maouche, Philadephia University, Amman
Louiza Bouallouche Medjkoune, LAMOS, Bejaia
Bruno Monsuez, ENSTA ParisTech, Saclay
Mohamed Mosbah, LaBRI, Bordeaux
Safia Nait-Bahloul, LITIO, Université d'Oran
Meriem Ouederni, IRIT, Toulouse
Claire Pagetti, ONERA, Toulouse
Vladimir Paun, ENSTA ParisTech, Palaiseau
Nihal Pekergin, LACL, Créteil
Olivier Perrin, INRIA-LORIA, Nancy
Denis Poitrenaud, LIP6, Paris
Florin Popentiu Vladicescu, Univ. of Oradea (**co-chair**)
Riadh Robbana, INSAT, Tunis
Zaïdi Sahnoun, LIRE, Constantine
Zohra Sbaï, ENIT, Tunis
Larbi Sekhri, Univ. es Senia, Oran
Nadia Tawbi, Université Laval, Quebec
Thouraya Tebibel, ESI, Alger
Ferucio Laurentiu Tiplea, University of Iasi
Tayssir Touili, LIPN Villetaneuse
Farouk Toumani, LIMOS, Clermont-Ferrand
Karsten Wolf, Universitaet Rostock

CONTENTS

I Session: Control and Diagnosis 7

Resilience Assessment: Accidental and Malicious Threats (***Invited talk***)
Mohamed KAÂNICHE 9

Fault Diagnosis of P-Time Labeled Petri Net Systems
Patrice BONHOMME 11

Combining Enumerative and Symbolic - Techniques for Diagnosis of Discrete-Event Systems
Abderraouf BOUSSIF, *Mohamed* GHAZEL *and Kais* KLAI 23

II Session: Program verification 35

Probabilistic Approaches for Time Critical Embedded Systems (***Invited talk***)
Liliana CUCU-GROSJEAN 37

Towards the Property-Based Testing of an L4 Microkernel API
Cosmin DRAGOMIR, *Lucian* MOGOSANU, *Mihai* CARABAS, *Razvan* DEACONESCU *and Nicolae* TAPUS 39

An Approach for Formal Verification of Updated Java Bytecode Programs
Razika LOUNAS, *Mohamed* MEZGHICHE *and Jean-Louis* LANET 51

State Space Reduction Strategie for Model Checking Concurrent C Programs
Amira METHNI, *Matthieu* LEMERRE, *Belgacem* BEN HEDIA, *Serge* HADDAD *and Kamel* BARKAOUI . 65

III Session: Performance evaluation 77

Timeout Interaction and Migration in Distributed Systems (***Invited talk***)
Gabriel CIOBANU 79

Model-Based Verification of the DMAMAC Protocol for Real-time Process Control
Admar Ajith Kumar SOMAPPA, *Andreas* PRINZ *and Lars* KRISTENSEN 81

On quantitative Analysis of Time Open Workflow Nets and Parametric Extension
Zohra SBAÏ *and Kamel* BARKAOUI 97

Verification of Bounded Real-Time Distributed Systems With Mobility
Bogdan AMAN *and Gabriel* CIOBANU 109

Part I

Session: Control and Diagnosis

Resilience Assessment: Accidental and Malicious Threats

Mohamed Kaâniche
CNRS; LAAS; Université de Toulouse – 7, Avenue du colonel Roche, F-31077 Toulouse, France
Université de Toulouse; UPS; INSA; INP; LAAS; F-31077 Toulouse, France
mohamed.kaaniche@laas.fr

A large body of research has been dedicated to the analysis, assessment and protection of cyber-physical systems and critical infrastructures against potential threats that might affect the dependability, the security or the resilience of the services delivered to the users. Traditionally, accidental and malicious threats have been taken into account separately. In this talk we will address the challenges raised by the resilience assessment and analysis of such systems considering accidental and malicious threats in an integrated way and we will present some examples of research studies carried out in this context.

Critical infrastructures, resilience, assessment, accidental threats, malicious threats

1. SUMMARY

In the past decade, several concerns have been raised about the vulnerability of critical infrastructures and cyber-physical systems and their efficient protection in the presence of accidental and malicious threats (Rahman et al. 2009).

Historically, most of the efforts were dedicated to the protection of critical infrastructures against accidental faults and natural disasters with a specific focus on safety. The situation changed significantly after the September 11, 2001 tragic events that led to increased international concerns about the security and robustness of critical infrastructures in response to evolving malicious threats

The vulnerability of critical infrastructures has increased as a result of the wider use of open networks and information infrastructures, and the proliferation of vulnerable operating systems and control devices. Recent events targeting critical infrastructures show that the threat is real. A widely reported example is the Stuxnet sophisticated malware discovered in July 2010 that targeted specific industrial computer control equipment and software, used for instance in nuclear power plants in Iran [(Langner 2011).

A large body of research has been dedicated to the analysis, assessment and protection of cyber-physical systems and critical infrastructures against potential threats that might affect the dependability, the security or the resilience of the services delivered to the users. The resilience term is used differently, by different communities. It is defined in (Laprie 2011) as the persistence of service delivery that can justifiably be trusted, when facing changes.

Traditionally, accidental and malicious threats have been taken into account separately. In this talk we will address the challenges raised by the resilience assessment and analysis of such systems considering accidental and malicious threats in an integrated way and we will present some examples of research studies carried out in this context.

In particular this objective has been addressed in the context of the CRUTIAL project (http://crutial.rse-web.it/) considering the example of power grid critical infrastructures and the associated information infrastructures dedicated to their management and control.

CRUTIAL focussed on the failures resulting from interdependencies between these infrastructures. The characterization of such failures and the modelling of their impact on relevant properties of power systems have been investigated by means of models at different abstraction levels: i) from a very abstract view expressing the essence of the typical phenomena due to the presence of interdependencies, ii) to an intermediate detail level representing in a rather abstract way the structure of the infrastructures, in some scenarios of interest, iii) to a quite detailed level where the infrastructures components and their interactions are investigated at a finer grain, considering elementary events

occurring at the components level and analysing their impact at the system level.

Accordingly, the proposed resilience assessment framework (Kaâniche et al. 2009) is based on a hierarchical modelling approach that accommodates the composition of different types of models and formalisms, including generalized stochastic Petri nets, fault trees, Stochastic Well formed Nets, and Stochastic Activity Networks. Additionally, a new formalism called “Dependent Automata” has been developed to provide a rigorous definition of interdependencies related failures. Also, unified models for describing cascading and escalating failures considering accidental and malicious threats in an integrated way have been defined (Laprie et al. 2007)

Besides these models, the CRUTIAL project resilience assessment activities included architecture validation activities as well as testbed based experiments to analyse the impact of different attack scenarios on control applications.

We will outline some of the results obtained in the context of this project and discuss some open research problems.

3. REFERENCES

Kaâniche, et al. (2009) CRUTIAL Project Deliverable D16 - Final version of the modelling framework. http://crutial.rse-web.it/Dissemination/DELIVERABLES-OF-THE-PROJECT.asp

Laprie, Jean-Claude, Kanoun, Karama, Mohamed Kaâniche, (2007) Modelling interdependencies between Electricity and Information Infrastructures. The 26th International Conference on Computer Safety, Reliability, and Security (SAFECOMP-2007), Nuremberg, Germany, LNCS 4680, Springer, pp. 54-67.

Laprie, Jean-Claude “From Dependability to Resilience”, IEEE International Conference on Dependable Systems and Networks (DSN-2008), Supplemental volume, Anchorage, Alaska, USA, pp. G8-G9, 2008.

Langner, R. “Stuxnet: Dissecting a Cyberwarfare Weapon,” *IEEE Security & Privacy*, vol. 9, no. 3, 2011, pp. 49–51.

Rahman, H.A., Beznosov, K., Marti, J.R., “Identification of sources of failures and their propagation in critical infrastructures from 12 years of public failure reports“, Int. Journal on Critical Infrastructures, vol.5, n°3, 2009

BIO

Mohamed Kaâniche has been at LAAS-CNRS, Toulouse, France, since 1988 where he currently holds a position of “Directeur de Recherche”, heading the Dependable Computing and Fault Tolerance Group. From March 1997 to February 1998, he was a Visiting Research Assistant Professor at the University of Illinois at Urbana-Champaign, IL, USA.

His research addresses the dependability and security assessment of hardware and software fault tolerant computer systems and critical infrastructures, using analytical modelling and experimental measurement techniques.

He has been involved in several national and European research projects and acted as a consultant for companies in France and as an expert for the European Commission. He has served on program and organization committees of international dependability related conferences. He was Program Chair of PRDC-2004, EDCC-5, DSN-PDS 2010, LADC-2011 and SAFECOMP- 2013. He is General co-Chair of DSN-2016 that will be held in Toulouse, France in June 2016.

Fault Diagnosis of P-Time Labeled Petri Net Systems

Patrice Bonhomme
University François-rabelais
CNRS, LI EA 6300, OC ERL CNRS 6305
64 avenue Jean Portalis
37200 Tours
France
bonhomme@univ-tours.fr

This paper focuses on the fault diagnosis problem of systems modeled with P-time labeled Petri nets with partial information. Indeed, the set of transitions is partitioned into those labeled with the empty string ϵ called silent (as their firin cannot be detected) including the faulty transitions and the observable ones. The proposed approach is based on the synthesis of a function called diagnoser allowing to determine the diagnosis state of the system based on the current observation. The novelty of the developed approach resides in the fact that, although the time factor is considered as intervals, the diagnoser is computed thanks to the underlying untimed Petri net structure of the P-time labeled model considered. Furthermore, the method relies on the schedulability analysis of particular firin sequences exhibited by the analysis of the obtained diagnoser and does not require the building of the state class graph.

Discrete event systems. Petri nets. Time labeled systems. Observability. State estimation. Fault diagnosis.

1. INTRODUCTION

The correct behavior of a real-world application is the ultimate requirement, particularly for systems such as communication protocols, manufacturing and real-time systems. Indeed, a drift from an expected behavior can be of crucial importance and can even lead, in extreme cases, to severe consequences including human losses. So, knowing the current state of a system in order to take the appropriate decisions and determining the malfunction of a system component are nowadays fundamental issues.

From a practical point of view, associating a dedicated sensor to each variable of interest in order to monitor its internal state is inconceivable. This restriction, due to economical or physical accessibility reasons leads to a system analysis in presence of uncertainties as the state information cannot be directly obtained. This particularity has gave rise to the introduction of the observers paradigm in the classical system theory. Indeed, an observer can be viewed as a mechanism allowing to estimate or reconstruct the internal state of a system based on some measurements. From a discrete event dynamic systems point of view and more precisely from a Petri net (PN on short) perspective this issue corresponds to the estimation of a PN marking based on some event observations.

Thus, being given a sequence of observed events (called word or trace) the challenge consists in determining if a fault has occurred, eventually or for sure!

It can be noticed that the problems of fault diagnosis has receive extensive attention these recent years and particularly in the framework of automata models and regular languages (Sampath et al. (1995), Cassandras and Lafortune (2008), Lin (1994), Cassez and Tripakis (2008)) but there are few studies in the time discrete event systems context.

A preliminary version of this paper was presented in (Bonhomme (2014)) where an approach allowing to estimate the marking of a P-time labeled Petri net (P-TLPN) system based on the observation of particular labels was presented. The plant observation is given by a set of labels whose occurrence can be detected/observed by an external agent (called observer or estimator) - these particular labels are associated to observable transitions. The other transitions, the unobservable ones (called silent transitions) are labeled with the empty string ϵ.

In this extended and enriched version, a fault diagnosis problem is solved thanks to the introduction of a function called diagnoser which associates to each observation a diagnosis state. In the proposed technique the set of unobservable transitions is further partitioned into the

set of faulty transitions and the set of regular ones. The regular transitions are unobservable and non faulty.

The proposed approach does not require the state class graph construction and consequently it is designed to alleviate the state space explosion problem. Indeed, the construction of the considered state observer is based on the analysis of the underlying untimed PN structure of the P-time labeled PN considered.

In particular, the following four assumptions are made:

1. the net structure and the initial marking are known,
2. the fault model is known,
3. the underlying untimed PN, of the P-TLPN considered is bounded,
4. the Petri net induced by the set of unobservable transitions does not contain circuit of null length.

Note that this latter assumption is adopted to exclude the situation where an infinit of actions may take place in a finit amount of time: it prevents the net induced by the set of unobservable transitions from being Zeno (Hadjidj et al. (2007)) which is in contradiction with a diagnosability scheme. In addition, there is no assumption on the backward conflic freeness of the subnet induced by the set of unobservable transitions as in (Giua et al. (2007)).

The paper is organized as follows: after an overview of the relevant literature in the next section, a brief reminder of the basics of untimed Petri nets followed by a formal definitio of P-time labeled Petri nets is realized in the third section. Section four covers the procedure of estimation and the construction of the state observer. The schedulability analysis of the occurrence sequence highlighted by the state observer and its application to the estimation problem are studied in the fift section. In the sixth section the fault diagnosis problem is solved. Section seven presents an illustration of the developed method and the last section concludes the paper and gives suggestions for future research.

2. LITERATURE REVIEW

For discrete event system (DES) state estimation has been addressed by several researchers. For instance, in (Giua et al. (2007)) the authors deal with the marking estimation of a labeled Petri net system. Thanks to structural assumptions on the subnet induced by the set of unobservable transitions, they propose an algebraic characterization of the set of consistent markings once a sequence is observed.

In the framework of fault detection or fault diagnosis several approaches can also be found in the literature - fault diagnosis is closed to the state estimation problem. Note that a complete survey of fault diagnosis methods for DES can be found in (Zaytoon and Lafortune (2013)). In (Cabasino et al. (2010)) the authors proposed a diagnosis approach based on the concept of basis marking and justificatio under the acyclicity assumption of the unobservable subnet of the system considered. Intuitively, for an observed sequence (word) ω, a justificatio can be thought as the set of minimal (in terms of firin vector) unobservable transitions interleaved with ω necessary to complete ω into a fireabl sequence on the net considered, from the initial marking. They extended their work in (Cabasino et al. (2014)) to provide a diagnosability approach for bounded labeled PN by introducing two graphs, namely the modifie basis reachability graph (MBRG) and the basis reachability diagnoser (obtained from the MBRG). Necessary and sufficien conditions for diagnosability are given but the construction of the two graphs is of exponential complexity with respect to the structure of the PN considered and its initial marking.

There are relatively few works in this topic in the time discrete event systems scheme where the time factor is modeled as intervals, so, numerous problems are still open. Concerning the time Petri net model of Merlin (Merlin and Faber (1976)), the authors in (Basile et al. (2013)) proposed a procedure for estimating the marking of the model in presence of unobservable transitions. They introduced a modifie state class graph which captures the required information on the possible evolution of the system starting from a given initial marking. Thanks to this graph, being given a timed sequence and a time instant, the set of markings consistent with the current observation is determined via integer linear programming techniques. The approach is restricted to bounded time Petri nets.

In a recent paper, the authors in (Basile et al. (2015)) extend the previously mentioned approach developed in (Basile et al. (2013)) to deal with the state estimation and the fault diagnosis problem for systems modeled by time PN augmented with labels.

The authors in (Wang et al. (2013)), thanks to a fault diagnosis graph (FDG) which is a truncation of the conventional state class graph (SCG) (Berthomieu and Diaz (1991)), developed an online technique for the fault diagnosis of systems modeled by unlabeled time Petri nets. The FDG is constructed incrementally with respect to the current observation and its number of states can be, in the worst case, the same as the one of the traditional state class graph. Indeed, the FDG is obtained from the SCG by only keeping the information required for the evaluation of the fault states and the authors concentrate on the sequence information and remove the irrelevant state classes (i.e., which are not used in the fault diagnosis). Intuitively, the state classes which are obtained after the firin of an unobservable transition are discarded as the diagnosis state is updated after an observation.

The acyclicity assumption of the subnet induced by the unobservable transitions is also considered. The authors further extend the method in (Wang et al. (2014)) by using reduction rules and model checking techniques.

3. PETRI NETS

3.1. Untimed Petri Nets

The reader unfamiliar with Petri nets can refer to (Murata (1989)), in the following only the basic notions are recalled.

A Place/Transition net (P/T net) is a structure $N = (P, T, Pre, Post)$, where P is a set of m places; T is a set of n transitions. $Pre : P \times T \to \mathbb{N}$ and $Post : P \times T \to \mathbb{N}$ are the pre and post incidence functions that specify the arcs; $C = Post - Pre$ is the incidence matrix. The preset and postset of a node $X \in P \cup T$ are denoted ${}^{\circ}X$ and X°. A marking is a vector $M : P \to \mathbb{N}$ that assigns to each place of a P/T net a non-negative integer number of tokens, represented by black dots. $M(p)$ is the marking of place p.

A net system $\langle N; M_0 \rangle$ is a net N with an initial marking M_0. A transition t is marking enabled at M if $M \geq Pre(\cdot, t)$. A transition t enabled at M may fire yielding the marking $M' = M + C(\cdot, t)$. We write $M[\sigma >$ to denote that the sequence of transitions σ is enabled at M, and we write $M[\sigma > M'$ to denote that the firin of σ yields M'. A marking M is reachable in $\langle N; M_0 \rangle$ iff there exists a firin sequence σ such that $M_0[\sigma > M$.

The set of all sequences that are enabled at the initial marking M_0 is denoted $L(N, M_0)$ i.e., $L(N, M_0) = \{\sigma \in T^{\star} | M_0[\sigma >\}$ with $T^{\star}$ the Kleene closure of set T i.e. the set of all firin sequences of elements of T of arbitrary length, including the empty sequence λ. The notation $\sigma'\sigma$ will correspond to the firin sequence σ' followed by firin sequence σ, i.e., the concatenation operation ; σ' is the prefi of firin sequence $\sigma'\sigma$.

The set of all markings reachable from M_0 define the reachability set of $\langle N; M_0 \rangle$ and is denoted $R(N, M_0)$.

Given a net $N = (P, T, Pre, Post)$ and a subset $T_s \subseteq T$, the T_s-induced subnet of N is the net $N_s = (P, T_s, Pre_s, Post_s)$ where Pre_s and $Post_s$ are the restrictions of Pre and $Post$ to T_s. So, the net N_s is obtained from N by removing all transitions in $T \setminus T_s$, it is denoted also by $N_s \angle_{T_s} N$.

3.2. Labels mapping

A labels mapping $\mathcal{LM}$ is associated to each transition of the net considered as follows

$$\mathcal{LM} : T \to \Omega \bigcup \{\epsilon\},$$

with Ω a finit alphabet and ϵ the empty string.

In the proposed approach, the set of transitions is partitioned into two sets: observable transitions whose firin can be detected by an external observer, denoted as T_o and unobservable transitions whose firin cannot be detected, denoted as T_u with $T = T_o \cup T_u$ and $T_o \cap T_u = \emptyset$.

More precisely, the following stands:

- $T_u = \{t \in T | \mathcal{LM}(t) = \epsilon\}$, transitions in T_u are also called silent,
- $T_o = \{t \in T | \mathcal{LM}(t) \neq \epsilon\}$ (i.e., T_o is the set of transitions labeled with a symbol in Ω).

In the proposed approach, the same label $\zeta \in \Omega$ can be shared by several transitions, i.e., two transitions t_i, t_j with $t_i \neq t_j$ will be called indistinguishable if:

$$\mathcal{LM}(t_i) = \mathcal{LM}(t_j) = \zeta.$$

The extension of the label mapping can be realized over sequences, $\mathcal{LM} : T^{\star} \to \Omega^{\star}$, recursively as follows:

1. $\mathcal{LM}(t_i) = \zeta \in \Omega$ if $t_i \in T_o$,
2. $\mathcal{LM}(t_i) = \epsilon$ if $t_i \in T_u$,
3. let $\sigma \in T^{\star}$ and $t_i \in T$ then $\mathcal{LM}(\sigma t_i) = \mathcal{LM}(\sigma)\mathcal{LM}(t_i)$,
4. $\mathcal{LM}(\lambda) = \epsilon$ where λ is the empty sequence.

3.3. P-time Petri Nets

Definitio 1 *The formal definitio of a P-TPN (Khansa et al. (1996)) is given by a pair $\langle N; I \rangle$ where:*

- *N is a marked Place/Transition net (a P/T net system augmented with a marking)*
- $P \to (\mathbb{Q}^{+} \cup \{0\}) \times (\mathbb{Q}^{+} \cup \{\infty\})$,
- $p_i \to I(p_i) = [a_i, b_i]$ *with* $0 \leq a_i \leq b_i$

With:

- P: the set of places of the net N,
- $\mathbb{Q}^{+}$: the set of positive rational numbers,
- I_i define the static interval of the operation duration of a token in a place p_i.

A token in place p_i will be considered in the enabledness of the output transitions of this place if it has stayed for a_i time units at least and b_i at the most. Consequently, the token must leave p_i, at the latest, when its operation duration becomes b_i. After this duration b_i, the token will be "dead" and will no longer be considered in the enabledness of the transitions. According to the strong firin mode, a transition in a P-TPN, is forced to fir unless it is disabled by the firin of another conflictin transition.

Let consider α_i the clock associated with the token denoted $i \in TK$ of the P-TPN (TK being the set of tokens of the P-TPN considered). υ is a valuation of the system, i.e., a mapping associating to each token i of the P-TPN, an element of $(\mathbb{R}_{\geq 0})$, υ_i, representing the time elapsed since the token i has been created (i.e., the valuation of the clock α_i). So, $\upsilon \in (\mathbb{R}_{\geq 0})^{TK}$ with the notation A^X representing the set of mappings from X to A. $\overline{0}$ is the initial valuation with $\forall i, \overline{0}_i = 0$

The semantics of a P-TPN can be define as a Timed Transition System (TTS). A state of the TTS is a couple $s = (M, \upsilon)$ where M is a marking and υ a valuation of the system.

Definitio 2 *The semantics of a P-TPN $\langle N; I\rangle$ is define by the Timed Transition System $\mathcal{S}_N = (\mathcal{Q}, \{q_0\}, \Sigma, \longrightarrow)$:*

1. $\mathcal{Q} = \mathbb{N}^P \times (\mathbb{Q}_{\geq 0})^{TK}$
2. $q_0 = (M_0, \overline{0})$
3. $\Sigma = T$
4. $\longrightarrow \in \mathcal{Q} \times (\Sigma \cup \mathbb{Q}_{\geq 0}) \times \mathcal{Q}$

- *The continuous transition is define $\forall d \in \mathbb{R}_{\geq 0}$ by:*

$$(M, \upsilon) \xrightarrow{d} (M, \upsilon') \text{ iff} \begin{cases} \upsilon' = \upsilon + d. \\ \forall \text{ token } k \text{ in } p_s \Rightarrow \upsilon'_k \leq b_s. \end{cases}$$

- *The discrete transition is define $\forall t_i \in T$ by:*

$(M, \upsilon) \xrightarrow{t_i} (M', \upsilon')$ *iff:*

$$\begin{cases} M \geq^{\circ} t_i. \\ \forall \text{ token } k \text{ in } p_l, \upsilon_k \leq b_l. \\ \forall p_s \in^{\circ} t_i, \forall \text{ token } k \text{ in } p_s \text{ involved in } t_i\text{'s firin :} \\ \bigcap_k [max(0, a_s - \upsilon_k), (b_s - \upsilon_k)] \neq \emptyset. \\ M' = M -^{\circ} t_i + t_i^{\circ}. \\ \forall \text{ token } r, \upsilon'_r = \begin{cases} 0 \text{ if created by } t_i. \\ \upsilon_r \text{ otherwise.} \end{cases} \end{cases}$$

The dynamic evolution of a P-TPN depends on the timing situation of each token. Indeed, each token will be associated with a potential firin interval (or dynamic interval) which can be different from its static one. For instance, consider place p_i with static interval $[a_i, b_i]$, let a token arrive in place p_i at absolute time τ. At τ its potential firin interval will correspond to $[a_i, b_i]$. At time $\tau + c$ with $c \leq b_i$ the dynamic interval of the considered token will become $[max(a_i - c, 0), b_i - c]$. It can be noticed that a token is considered as dead when its dynamic interval becomes $[0, 0]$.

Definitio 3 *A P-time labeled Petri net (P-TLPN on short) over an alphabet Ω is a triple $\langle N, I, \mathcal{LM}\rangle$ where $\langle N, I\rangle$ is a P-TPN and $\mathcal{LM} : T \to \Omega \bigcup \{\epsilon\}$ is a labeling function.*

Finally, given a sequence of labels (a word) $\omega \in \Omega^{\star}$, it is denoted by ω^k the kth element in ω and the number of elements of ω is denoted by $|\omega|$. For $a \in \Omega$, we write $a \in \omega$ if there exists $k \geq 1$ such that $\omega^k = a$ (i.e., a is an element of the word ω).

Furthermore, let $\omega_1, \omega_2, \ldots, \omega_n$ be n sequences of labels (i.e., $w_i \in \Omega^{\star}, 1 \leq i \leq n$), the notation $\omega = \omega_1\omega_2 \ldots \omega_n$ will be the concatenation of $\omega_1, \omega_2, \ldots, \omega_n$.

The next section recalls the procedure (Bonhomme (2015)) to construct the state observer.

4. ESTIMATION PROCEDURE

The goal of the observer is to give the current state estimate of the system based on the information of the observed traces. The state of the observer will consist in a set of states the model can be in after a label observation.

The following set will be associated to any observed word ω (i.e., the observed labels sequence):

- $\mathcal{L}(\omega)$ is the set containing all sequences of transitions that are consistent with ω, i.e., the set of all possible firin sequences that produce observation ω.

In general, if ω is an observed word, the associated firin sequence $\sigma \in \mathcal{LM}^{-1}(\omega)$ is not necessarily fireabl on the net as some unobservable transitions should be interleaved to obtain a fireabl sequence that produce ω.

Definitio 4 *Let N be a P-TLPN with $T = T_o \cup T_u$. The following operator is defined*

- *The projection over T_o is $P_o : T^{\star} \to T_o^{\star}$ define as:*
  - $P_o(\lambda) = \lambda$,
  - *for all $\sigma \in T^{\star}$ and $t \in T$, $P_o(\sigma t) = P_o(\sigma)t$ if $t \in T_o$ and $P_o(\sigma t) = P_o(\sigma)$ otherwise (with λ representing the empty sequence).*

Given a sequence $\sigma \in L(N, M_0)$, $\omega = \mathcal{LM}(P_o(\sigma))$ denotes the corresponding observed word.

Definitio 5 *Let N be a P-TLPN with $T = T_o \cup T_u$ and $\omega \in \Omega^{\star}$ be an observed word. $\mathcal{L}(\omega)$ is define as:*

$$\mathcal{L}(\omega) = P_o^{-1}(\mathcal{LM}^{-1}(\omega)) \cap L(N, M_0) = \{\sigma \in L(N, M_0) | \mathcal{LM}(P_o(\sigma)) = \omega\},$$

i.e., the set of firin sequences consistent with $\omega \in \Omega^{\star}$.

Definitio 6 *Let N be a P-TLPN with $T = T_o \cup T_u$ and $\omega \in \Omega^{\star}$ be an observed word. $\mathcal{C}(\omega)$ is define as:*

$$\mathcal{C}(\omega) = \{M \in R(N, M_0) | \exists \sigma \in \mathcal{L}(\omega) : M_0[\sigma > M\},$$

i.e., the set of markings consistent with ω.

So, being given an observed word ω, $\mathcal{L}(\omega)$ is the set of sequences that may have fire while $\mathcal{C}(\omega)$ is the set of markings in which the system may actually be.

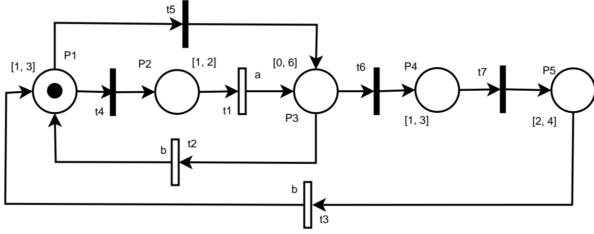


Figure 1: *P-TLPN model.*

Let consider the P-TLPN of Figure 1 with $T_u = \{t_4, t_5, t_6, t_7\}$, $T_o = \{t_1, t_2, t_3\}$, $\Omega = \{a, b\}$. It holds $\mathcal{LM}(t_1) = a, \mathcal{LM}(t_2) = \mathcal{LM}(t_3) = b$ (transitions t_2 and t_3 are indistinguishable) and $\mathcal{LM}(t_i) = \epsilon, \forall t_i \in T_u$.

If the observed word is $\omega = ab$ then $\mathcal{LM}^{-1}(\omega) = \{t_1t_2, t_1t_3\}$ and $\mathcal{L}(\omega) = \{t_4t_1t_2, t_4t_1t_6t_7t_3\}$ and $\mathcal{C}(\omega) =$ [10000].

Definitio 7 *Let N be a P-TLPN with $T = T_o \cup T_u$, the unobservable reachability mapping $\mathcal{UR}$, which enables to fin the markings reachable from a given marking M_i, following the firin of all unobservable sequences is define as:*

$\mathcal{UR} : \mathbb{N}^m \to 2^{\mathbb{N}^m}$,

$M_i \to \mathcal{UR}(M_i) = \{M_j \in \mathbb{N}^m | \exists \sigma_u \in T_u^\star, M_i[\sigma_u > M_j\}$,

with $2^{\mathbb{N}^m}$ the power set of the markings of the PN considered.

4.1. State observer

Let N_i and N_j be two nodes of the graphical representation of the state observer (associated respectively to the states y_i and y_j of the observer) such that it exists a directed arc linking N_i to N_j ($N_i \to N_j$, i.e., N_i is a predecessor of N_j) labeled with a_k with $a_k \in \Omega$ as illustrated on Figure2.

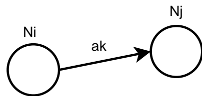


Figure 2: *nodes of the state observer.*

Definitio 8 *The state observer for the partially observable P-TLPN N with initial marking M_0 and $T = T_o \cup T_u$ is define by the 5-tuple $(Y_{so}, E_{so}, f_{so}, y_0, \varsigma_{so})$ where:*

- *Y_{so} is the set of states of the state observer,*
- *$E_{so} = \Omega$ is the set of labels (associated to the observable events),*
- *$\varsigma_{so} : Y_{so} \to 2^{R(N,M_0)}$ is a function associating to each state $y_{so} \in Y_{so}$ a set of reachable markings,*
- *y_0 is the initial state of the state observer and $\varsigma_{so}(y_0) = SEM(N_0) \cup SSM(N_0)$,*
- *$f_{so} : Y_{so} \times E_{so}^\star \to Y_{so}$ is the transition function define as :*
  for $y_l \in Y_{so}$ a state of the observer and $\omega \in E_{so}^\star$ a string of observable labels $f_{so}(y_0, \omega) = y_l$ if $\varsigma_{so}(y_l) \notin \emptyset$ where $\varsigma_{so}(y_l) = \left\{M_l : M_0 \xrightarrow{\tau} M_l \wedge \mathcal{LM}(P_o(\tau)) = \omega\right\} = SEM(N_l) \cup SSM(N_l)$.

With the two sets SSM and SEM define as follows:

Definitio 9 *Sets SSM and SEM*

- *$SEM(N_j)$, the Set of Entry Markings of N_j,*

$$SEM(N_j) = \{M_s \in N_j | \exists M_u \in N_i, t_k \in T_o, \\ a_k \in \Omega, \mathcal{LM}(t_k) = a_k : M_u[t_k > M_s\}$$

- *$SSM(N_j)$, the Set of Shadow Markings of N_j,*

$$SSM(N_j) = \{M_s \in N_j | \exists M_u \in SEM(N_j), \\ \sigma_u \in T_u^\star : M_u[\sigma_u > M_s\}$$

or equivalently, $SSM(N_j) = \mathcal{UR}(SEM(N_j))$.

Intuitively, for a given node N_s of the state observer, after the observation of the word ω, the set $SEM(N_s) \cup SSM(N_s)$ represents the set of markings that are consistent with the current observed word (i.e., $\mathcal{C}(\omega)$). The other nodes can be computed recursively as explained in the following.

1. The state observer starts in the initial state y_0 and its associated initial node N_0 is composed of $SEM(N_0) = \{M_0\}$ and $SSM(N_0) = \mathcal{UR}(M_0)$.
2. as soon as a label a_k (associated with an observable transition $t_k \in T_o$) is observed a new state y_l of the observer is calculated yielding a new node N_l:
   - the set of entry markings of node N_l is obtained by investigating the set of markings resulting from the firin of transition t_k starting from any marking ($SEM \cup SSM$) of N_0,
   - the set of shadow markings of N_l corresponds to the set of markings obtained by the firin of all unobservable sequences of transitions starting from any entry marking of N_l,

3. return to 2 with the newly calculated state as the initial state.

Definitio 10 *Let N_i and N_j be two nodes of the state observer, N_i and N_j are said to be equivalent ($N_i \Leftrightarrow N_j$) if and only if:*

$SEM(N_i) = SEM(N_j)$ *and* $SSM(N_i) = SSM(N_j)$.

Proposition 1 *Two nodes N_i and N_j of the state observer will be equivalent if and only if, the following holds:*

$SEM(N_i) = SEM(N_j)$.

Definitio 11 *Given a marking $M_i \in R(N, M_0)$ and a transition $t_f \in T_o$ (associated with a label $l_f \in \Omega$, i.e., $\mathcal{LM}(t_f) = l_f$), the set of candidate sequences denoted $CS(M_i, t_f)$ is the set of firin sequences, composed of the unique fina observable transition t_f, which can occur from M_i, i.e.:*

$CS(M_i, t_f) = \{s.t_f | s \in T_u^{\star} \cup \lambda, t_f \in T_o : M_i[s.t_f >\}$.

With respect to the timing constraints to be satisfied candidate sequences can be in the state possible or impossible.

As $N_u \angle_{T_u} N$ (i.e., the Petri net induced by the set of unobservable transitions) is not Zeno by assumption, it is ensured that the time is diverging with regard to the length of the firin sequences, thus, the set of candidate sequences from a marking is necessarily finit (at the instant of observation) and it can be investigated. The following section addresses the schedulability analysis (Bonhomme (2013b)) of an occurrence sequence (i.e., a procedure verifying if the considered firin sequence can occur without any violation of timing constraints) and its application to the estimation problem.

5. SCHEDULABILITY ANALYSIS AND ESTIMATION

Let $\sigma = t_a t_b t_c \ldots t_q$ be a firin sequence of length s (denoted $|\sigma| = s$). The jth fire transition of σ will be associated with the jth firin instant (Bonhomme (2013a)). A variable x_i will represent the elapsed time between the $(i-1)^{th}$ firin instant and the ith one (with $x_0 = 0$).

For instance on Figure 3, $(x_2 + x_3)$ is the time elapsed between the firs firin instant (associated with transition t_a) and the third one (transition t_c).

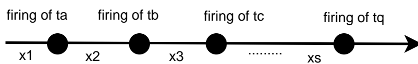


Figure 3: *Firing instants.*

In a P-TPN, the sojourn time (i.e., the amount of time that a token has been waiting in a place) is counted up as soon as the token has been dropped in the place as seen previously. To compute the firin instants, this approach requires that a token is identifie by three parameters: the place that contains it, the information of its creation instant and of its consumption one.

Function TOK is define with this purpose assuming that a FIFO queuing policy in the net is used in the sequel:

TOK:$\mathbb{N} \times (\mathbb{N} \setminus \{0\}) \times T^{\star} \rightarrow \wp(P))$,

$TOK(j, n, \sigma) = \{p \in P | p$ contains a token created by the jth firin instant and consumed by the nth one in firin sequence $\sigma\}$.

With $\wp(P)$ the set of subsets of P (also noted 2^P).

When it is clear from the context σ will be omitted in the notation of $TOK(.)$.

When the weight of the P-TPN arcs is element of $\mathbb{N}$, $TOK(j, n)$ is a multi-set. For the sake of simplicity, only ordinary P-TPN are considered (the arcs weight are element of $\{0, 1\}$).

Tokens, with the same creation instant, located in different places and involved in the same transition firin may mutually constrained their sojourn time, the following quantities, $Dsmin$ and $Dsmax$, are introduced in order to evaluate the contribution of these tokens. So, $Dsmin$ represents their availability in order to participate to this firin and similarly, $Dsmax$ expresses the fact that they all must be prevented from dying (with $[a_i, b_i]$ the static interval associated with the place p_i).

$$Dsmin(j,n) = \begin{cases} \max(a_i), & i \mid p_i \in TOK(j,n) \\ \text{else } 0 \text{ if } TOK(j,n) = \emptyset \end{cases},$$

$$Dsmax(j,n) = \begin{cases} \min(b_i), & i \mid p_i \in TOK(j,n) \\ \text{else } +\infty \text{ if } TOK(j,n) = \emptyset \end{cases}.$$

The definitio of the following set $SEN(q)$, allowing to determine the creation instants of tokens involved in the qth firin instant, is also necessary:

$$SEN(q) = \{u | TOK(u, q) \subset ({}^{\circ}t_q)\}$$

To express more simply the obtained results, the definitio of the following coefficient is required:

$$c_{uq} = \begin{cases} Dsmin(u,q) & \text{if } u \in SEN(q) \\ 0 & \text{else} \end{cases},$$

$$d_{jk} = \begin{cases} Dsmax(j,k) & \text{if } TOK(j,k) \neq \emptyset \\ +\infty & \text{else} \end{cases}$$

With, $\forall (j,k) \in [0, q-1] \times [1, q], j \notin SEN(q)$ and $k \neq q$, then $c_{jk} = 0$, and $\forall k \in [0, q], x_k \geq 0$.

The following proposition is finall obtained:

Proposition 2 *A sequence of transitions $\sigma = t_1 t_2 ....t_q$ is schedulable (i.e., it may be fi ed respectively at firin instants $1, 2, \ldots, q$) if and only if there exist $x_1 \geq 0$, $x_2 \geq 0$,..., $x_q \geq 0$ such that:*

$$\begin{cases} c_{0k} \leq x_1 \leq d_{0k}, k = 1, ..., n \\ \max\limits_{k=2,\ldots,n} (c_{0k}, c_{1k} + x_1) \leq x_1 + x_2 \leq \min\limits_{k=2,\ldots,n} (d_{0k}, d_{1k} + x_1) \\ \ldots \\ \max\limits_{\substack{j=0,\ldots,q-1 \\ k=q,\ldots,n}} (c_{jk} + \sum\limits_{s=0}^{j} x_s) \leq \sum\limits_{s=0}^{q} x_s \leq \min\limits_{\substack{j=0,\ldots,q-1 \\ k=q,\ldots,n}} (d_{jk} + \sum\limits_{s=0}^{j} x_s) \end{cases}$$

In the sequel this system will be denoted as $\mathcal{S}_\sigma(q)$ or simply $\mathcal{S}_\sigma$ when it is clear from the context.

Definitio 12 *The firin space at the qth firin instant, associated with a firin sequence σ, denoted by $\mathcal{FS}_\sigma(q)$ is the set of non negative vectors $(x_1, ..., x_q)$ such that the fi st, the second, ... and the qth firin conditions are satisfied Thus, a firin sequence $\sigma = t_1 t_2 ....t_q$ is schedulable if and only if its associated firin space $\mathcal{FS}_\sigma(q)$ is non-empty.*

Thanks to this characterization of a firin sequence, the Zenoness property can be checked by evaluating the minimal duration of the circuit of unobservable transitions under consideration (for instance, by minimizing the sum of the x_i associated with the considered transitions).

Definitio 13 *A P-TLPN N_r firin schedule, will be a sequence of ordered pairs $(t_i, \sum\limits_{k=0}^{i} x_k)$; transition t_i fi able at time $(\sum\limits_{k=0}^{i} x_k)$, obtained from the state reached by starting from N_r initial state and firin the transitions t_j, $1 \leq j < i$, in the schedule at the given times.*

Finally, as in (Basile et al. (2015)), let denote:

$\omega_t = ((a_1, \tau_1), (a_2, \tau_2) \ldots (a_n, \tau_n)) \in (\Omega \times \mathbb{Q}^+)^\star$,

a time-label sequence (TLS), i.e., a sequence of pairs (observed label-time instant).

Indeed, in the considered sequence, label a_i is observed at absolute time τ_i ($i \geq 1$) and $\tau_1 \leq \tau_2 \ldots \leq \tau_n$.

Now all the required material for the proposed method is given, the principle is presented as follows:

- starting from the initial state, once a label a_f will be observed at the absolute time τ_f,
- the set of associated observable event $T_{a_f} = \{t \in T_o | \mathcal{LM}(t) = a_f\}$ will be evaluated,
- then, $\forall t_f \in T_{a_f}$ the set of feasible candidate sequences $CS(M_0, t_f)$ will be computed,
- a switch from node N_0 to node N_f (created by the observation of label a_f) is realized in the state observer,
- for each $\sigma_f \in CS(M_0, t_f)$ (with $P_o(\sigma_f) = t_f$) the associated linear system $\mathcal{S}_{\sigma_f}$ will be constructed,
- and each σ_f will be checked for schedulability with the following additional constraint:

$$\textstyle\sum_{i=0}^{|\sigma_f|} x_i = \tau_f.$$

Thanks to these considerations it is ensured that sequence σ_f is schedulable and the firin of t_f occurs at τ_f. Once a firin sequence is proved to be possible the set of markings the system can be in is then determined.

Let denote by $FEAS(N_0, t_f)$ the set of schedulable firin sequences from node N_0 ending with the unique observable transition t_f (it is a subset of the set of candidate sequences).

$$FEAS(N_0, t_f) = \Big\{\sigma \in CS(M_0, t_f) | \mathcal{FS}_\sigma(|\sigma|) \text{ augmented with } \textstyle\sum_{i=0}^{|\sigma|} x_i = \tau_f \text{ is non-empty}\Big\}.$$

Furthermore, based on the knowledge of the schedulable candidate firin sequences only a subset of the set of entry markings of node N_f (resulting from the firin of transition t_f), denoted $SEM'(N_f)$, will be considered for the next step.
It holds:

$$SEM'(N_f) = \{M \in SEM(N_f) | M_0[\sigma > M, \sigma \in FEAS(N_0, t_f)\}.$$

With $SEM'(N_f) \subseteq SEM(N_f)$.

Afterwards, if another label a_x is observed at absolute time τ_x then:

- The set of associated observable event $T_{a_x} = \{t \in T_o | \mathcal{LM}(t) = a_x\}$ will be evaluated,
- then, $\forall t_x \in T_{a_x}$ the set of feasible candidate sequences $CS(M_i, t_x)$ will be computed with $M_i \in SEM'(N_f)$,
- a switch from node N_f to node N_x is realized in the state observer,
- for each feasible firin sequence (on the underlying untimed PN) $\sigma'_f \sigma_x$ (i.e., $M_0[\sigma'_f \sigma_x >$) with $\sigma_x \in CS(M_i, t_x)$ and $\sigma'_f \in FEAS(N_0, t_f)$ the associated linear system $\mathcal{S}_{\sigma'_f \sigma_x}$ will be constructed.

  It is recalled that σ'_f is a schedulable firin sequence determined in the previous step with label a_f observed at τ_f and $P_o(\sigma'_f \sigma_x) = t_f t_x$.
- each previously determined $\sigma'_f \sigma_x$ will be checked for schedulability with the following additional constraint:

$$\textstyle\sum_{i=0}^{|\sigma'_f|+|\sigma_x|} x_i = \tau_x.$$

ensuring that the firin of t_x occurs at τ_x.

And so on, the same method is iteratively applied with respect to the current observation.

So, more formally the following principle is obtained: let ω_{obs} be an observed word (i.e., a sequence of labels $\omega_{obs} = a_1a_2a_3\ldots a_ia_{i+1}\ldots \in \Omega^\star$) and let N_i $(i \geq 1)$ be the node of the associated state observer obtained after the observation of label $a_i \in \omega_{obs}$ detected at absolute time τ_i, as illustrated on the following figur (Figure 4).

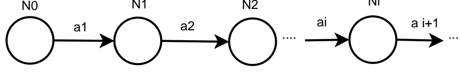


Figure 4: *Observable sequence.*

The associated sets $FEAS$ and SEM' are computed as follows:

Let $t_1 \in T_{a_1} = \{t \in T_o | \mathcal{LM}(t) = a_1\}$,

$FEAS(N_0, t_1) = \{\sigma \in CS(M_0, t_1) | \mathcal{FS}_\sigma(|\sigma|)$ augmented with $\sum_{k=0}^{|\sigma|} x_k = \tau_1$ is non-empty$\Big\}$.

$SEM'(N_0) = SEM(N_0) = \{M_0\}$ and

$$SEM'(N_1) = \{M \in SEM(N_1) | M_0[\sigma > M, \sigma \in FEAS(N_0, t_1)\}.$$

$\forall i > 0$,

Let $\mathcal{LM}(t_{i+1}) = a_{i+1}$,

$FEAS(N_i, t_{i+1}) = \{\sigma \in CS(M_b, t_{i+1}) | M_b \in SEM'(N_i)$, $M_0[\varpi >, \mathcal{FS}_\varpi(|\varpi|)$ augmented with $\sum_{k=0}^{|\varpi|} x_k = \tau_{i+1}$ is non-empty$\Big\}$.

With firin sequence $\varpi = \sigma_1\sigma_2\ldots\sigma_i\sigma$ where $\sigma_s \in FEAS(N_{s-1}, t_s), s \in \{1, \ldots, i\}$ and $P_o(\varpi) = t_1t_2t_3\ldots t_it_{i+1}$.

More precisely:

$P_o(\sigma_j) = t_j, j \in \{1, \ldots, i\}$ with $\mathcal{LM}(t_j) = a_j$.

$$SEM'(N_{i+1}) = \{M \in SEM(N_{i+1}) | M_k[\sigma > M, \sigma \in FEAS(N_i, t_{i+1}), M_k \in SEM(N_i)\}.$$

$SEM'(N_i)$ is the set of entry markings of node N_i resulting from the firin of schedulable firin sequences with respect to the current observation.

Roughly speaking, $FEAS(N_i, t_k)$ is the set of candidate sequences of node N_i ending with t_k and which can be completed by schedulable sub-sequences into a schedulable firin sequence starting from the initial marking of the P-TLPN considered.

So, by this way it is ensured that the feasible firin sequences associated with the observed time-label sequence $((a_1, \tau_1), (a_2, \tau_2)\ldots(a_{i+1}, \tau_{i+1}))$ are effectively computed.

In the next section, addressing the fault diagnosis problem of a P-TLPN system, this set will be used to evaluate the state diagnosis associated with an observed TLS.

6. FAULT DIAGNOSIS

The set of unobservable transitions is partitioned into two subsets, $T_u = T_f \cup T_{reg}$ where the set T_f includes all the fault transitions (modeling anomalous or faulty behavior) while T_{reg} includes all unobservable transitions which correspond to regular events. Furthermore, the set T_f is partitioned into r different subsets T_f^i, where $i = 1, \ldots, r$, that models the different fault classes.

Definitio 14 *Let $\langle N; M_0\rangle$ be a net system with labeling function $\mathcal{LM} : T \rightarrow \Omega \bigcup \{\epsilon\}$, where $N = (P, T, Pre, Post)$ and $T = T_o \cup T_u$. Let consider the TLS $\omega_t = ((a_1, \tau_1), (a_2, \tau_2)\ldots(a_n, \tau_n))$ associated with the state observer of Figure 4.*

Let define

$\sum(M_0, \omega_t) = \{\sigma \in T^\star | M_0[\sigma >, \sigma = \sigma_1\sigma_2\ldots\sigma_n :$

$\mathcal{LM}(\sigma_i) = a_i, i = 1, \ldots, n, \sigma_s \in FEAS(N_{s-1}, t_s),$

$\mathcal{LM}(t_s) = a_s, s = 1, \ldots, n\}$

Indeed, σ can be viewed as a concatenation of subsequences, namely $\sigma_i, i \geq 1$. Each subsequence σ_i is of the form $s.t_i$ with $s \in T_u^\star$, $\mathcal{LM}(t_i) = a_i$ and absolute firin instant of t_i is τ_i.

So, it holds:

$\sigma_i \in CS(M_b, t_i)$ with $M_b \in SEM'(N_{i-1})$.

Definitio 15 *A diagnoser is a function*

$$\Gamma : [\Omega \times \mathbb{Q}^+]^\star \times \{T_f^1, T_f^2, \ldots, T_f^r\} \rightarrow \{N, U, F\}$$

that associates with each observed time-label sequence ω_t and each fault class T_f^i, where $i = 1, \ldots, r$, a diagnosis state.

- $\Gamma(\omega_t, T_f^i) = N$ if $\forall\sigma \in \sum(M_0, \omega_t)$ and $\forall t_f \in T_f^i$, it is $t_f \notin \sigma$.

  In such a case the ith fault cannot have occurred, because none of the firin sequences consistent

with the considered observation contains a fault transition of class i.

- $\Gamma(\omega_t, T_f^i) = U$ if:
  1. $\exists\sigma \in \sum(M_0, \omega_t)$ and $t_f \in T_f^i$ such that $t_f \in \sigma$,
  2. $\exists\sigma' \in \sum(M_0, \omega_t)$ such that $\forall t_f \in T_f^i$, it is $t_f \notin \sigma'$.

  In such a case a fault transition of class i may have occurred or not, the diagnosis is in this case, uncertain.

- $\Gamma(\omega_t, T_f^i) = F$ if $\forall\sigma \in \sum(M_0, \omega_t), \exists t_f \in T_f^i$ such that $t_f \in \sigma$.

  In such a case the fault of class i must have occurred, because all firabl sequences consistent with the considered observation contains at least one fault transition of class i.

Let consider the P-TLPN of Figure1 with $T_u = \{t_4, t_5, t_6, t_7\}$, $T_o = \{t_1, t_2, t_3\}$, $\Omega = \{a, b\}$. It holds $\mathcal{LM}(t_1) = a, \mathcal{LM}(t_2) = \mathcal{LM}(t_3) = b$ (transitions t_2 and t_3 are indistinguishable). Furthermore, $T_f^1 = \{t_5\}$ and $T_f^2 = \{t_7\}$, i.e., there are two fault classes.

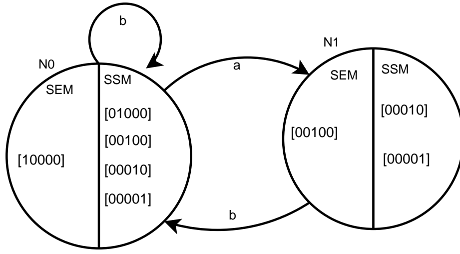


Figure 5: *State observer.*

The corresponding state observer with two nodes is depicted on Figure 5.

Let consider the following observed TLS $\omega_t = ((a, 2), (b, 5))$ then:

$\sum(M_0, \omega_t) = \{\omega_1, \omega_2\}$ with $\omega_1 = t_4t_1t_2$ and $\omega_2 = t_4t_1t_6\underline{t_7}t_3$.

We have (according to the notations of definitio 14):

- $\omega_1 = \sigma_1\sigma_2$ with $\sigma_1 = t_4t_1$ and $\sigma_2 = t_2$,
- $\omega_2 = \sigma_1\sigma_2$ with $\sigma_1 = t_4t_1$ and $\sigma_2 = t_6\underline{t_7}t_3$.

The two obtained candidate sequences are feasible with regard to the timing constraints. Indeed, the two associated firin schedules can be, for instance, considered respectively for ω_1 and ω_2:

- $((t_4, 1), (t_1, 2), (t_2, 5))$,
- $((t_4, 1), (t_1, 2), (t_6, 2), (t_7, 3), (t_3, 5))$.

It holds $t_7 \in T_f^2$ and $t_7 \in \omega_2$ ($t_7 \notin \omega_1$), and $t_5 \in T_f^1$, $t_5 \notin \omega_1, t_5 \notin \omega_2$.

So, $\Gamma(\omega_t, T_f^1) = N$ and $\Gamma(\omega_t, T_f^2) = U$.

It means, that according to the previous observed time label sequence ω_t, it is known for sure that the fault of class 1 (corresponding to fault transition t_5) cannot have occurred while fault transition $t_7 \in T_f^2$ may have occurred (via ω_2).

If the observed TLS corresponds to $\omega_t = (b, 1)$, it is easy to verify that $\sum(M_0, \omega_t) = \{\omega_3\}$ with $\omega_3 = t_5t_2$ (the associated firin schedule is $((t_5, 1), (t_2, 1))$) and consequently, $\Gamma(\omega_t, T_f^1) = F$ and $\Gamma(\omega_t, T_f^2) = N$ (i.e., a fault of class T_f^1 occurs for sure and a fault of the second class cannot have occurred).

In the next section an illustrative example is presented where the T_u-induced subnet is cyclic.

7. ILLUSTRATIVE EXAMPLE

Let consider the P-TLPN of Figure 6 with $T_o = \{t_2, t_5\}$, $T_u = \{t_1, t_3, t_4, t_6, t_7\}$, $T_f = \{t_6\}$ and $\mathcal{LM}(t_2) = a, \mathcal{LM}(t_5) = b$. The T_u-induced subnet contains the cycle $(p_3 - t_4 - p_4 - t_6 - p_3)$.

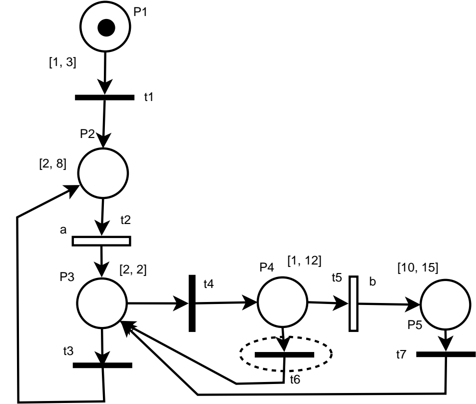


Figure 6: *P-TLPN with a cyclic T_u-induced subnet.*

The state observer is depicted on Figure 7, it consists of three nodes X_0, X_1 and X_2.

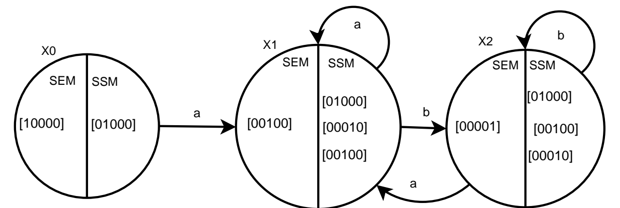


Figure 7: *State observer of the P-TLPN of Figure 6.*

If the observed word is $\omega = (a, b)$ then the set of possible associated firin sequences is of the form $t_1t_2t_4(t_6t_4)^\star t_5$ with the $\star$ after the subsequence (t_6t_4) (derived from the

Kleene star operator) indicating that it is allowed to occur from zero time to infinitel . Thanks to the time instant of occurrence of each label the set of feasible associated firin sequences is necessarily finite

For instance if the TLS considered is:

$\omega_t = ((a,3),(b,6))$ then $\sum(M_0,\omega_t) = \{\omega_1\}$ with $\omega_1 = t_1t_2t_4t_5$. The associated firin space $\mathcal{FS}_{\omega_1}(|\omega_1|)$ augmented with the following constraints:

- $x_1+x_2 = 3$ (absolute firin instant of transition t_2),
- $x_1+x_2+x_3+x_4 = 6$ (absolute firin instant of transition t_5),

is non-empty.

It holds:

$\omega_1 = \sigma_1\sigma_2$ with $\sigma_1 = t_1t_2$ and $\sigma_2 = t_4t_5$ and an example of firin schedule is:

$\varpi = ((t_1,1),(t_2,3),(t_4,5),(t_5,6))$,

and it is unique with respect to the static intervals of the P-TLPN places. So, it is easy to see that $\Gamma(\omega_t,T_f) = N$ and the faulty transition t_6 cannot have occurred.

If the TLS considered is now: $\omega_t = ((a,3),(b,9))$ then $\Gamma(\omega_t,T_f) = U$, as the computation of the set $\sum(M_0,\omega_t)$ leads to the following possible firin schedules (with the same observable projection), one containing the faulty transition and the other one not:

- $\varpi_1 = ((t_1,1),(t_2,3),(t_4,5),(t_5,9))$,
- $\varpi_2 = ((t_1,1),(t_2,3),(t_4,5),(\underline{t_6},6),(t_4,8),(t_5,9))$.

If the TLS considered is now: $\omega_t = ((a,3),(a,14))$ then $\Gamma(\omega_t,T_f) = F$. Indeed, the computation of the set $\sum(M_0,\omega_t)$ leads to the following possible firin schedule containing the faulty transition:

- $\varpi_2 = ((t_1,1),(t_2,3),(t_4,5),(\underline{t_6},10),(t_3,12),(t_2,14))$.

In this case the faulty transition occurs with certainty thanks to the timing structure of the P-TLPN considered and the occurrence date of the observed labels.

8. CONCLUSION AND PERSPECTIVES

In this paper, a new methodology allowing to analyze the fault diagnosis of systems modeled by P-time labeled Petri nets is developed. It is based on the construction of a function called diagnoser which associates with each observation and each fault class a diagnosis state. This diagnoser is obtained thanks to the synthesis of a state observer which is an automaton allowing to estimate the set of markings in which the system may be, being given a sequence of observed labels.

Furthermore, the considered state observer is computed on the basis of the untimed underlying Petri net of the P-time labeled PN considered. This particularity allows to avoid the combinatorial state space explosion problem usually associated with the consideration of the time factor modeled as time intervals.

Thanks to a schedulability analysis technique, the feasibility of the candidate firin sequences associated with the observed time-label sequence is evaluated via linear programming techniques.

An issue currently being investigated is the extension of the method to test the diagnosability property of P-TLPN systems, i.e., is the fault can be detected within a finit number of steps after its occurrence ?

REFERENCES

Basile, F., M. Cabasino, and C. Seatzu (2015, April). State estimation and fault diagnosis of labeled time petri net systems with unobservable transitions. *Automatic Control, IEEE Transactions on 60*(4), 997–1009.

Basile, F., M. P. Cabasino, and C. Seatzu (2013). Marking estimation of time Petri nets with unobservable transitions. In *IEEE Emerging Technologies and Factory Automation (ETFA)*, pp. 1–7.

Berthomieu, B. and M. Diaz (1991, March). Modeling and verificatio of time dependent systems using time petri nets. *IEEE Trans. Softw. Eng. 17*(3), 259–273.

Bonhomme, P. (2013a). Scheduling and control of real-time systems based on a token player approach. *Journal of Discrete Event Dynamic Systems 23*(2), 197–209.

Bonhomme, P. (2013b). Towards a new schedulability technique of real-time systems modeled by p-time Petri nets. *International Journal of Advanced Manufacturing Technology 67*(1-4), 759–769.

Bonhomme, P. (2014). Estimation of p-time labeled petri nets with unobservable transitions. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation, ETFA 2014, Barcelona, Spain, September 16-19, 2014*, pp. 1–8.

Bonhomme, P. (2015). Marking estimation of P-time Petri nets with unobservable transitions. *IEEE Transactions on Systems, Man, and Cybernetics: Systems 45*(3), 508–518.

Cabasino, M., A. Giua, and C. Seatzu (2010). Fault detection for discrete event systems using Petri nets with unobservable transitions. *Automatica 46*(9), 1531–1539.

Cabasino, M. P., A. Giua, and C. Seatzu (2014). Diagnosability of discrete event systems using labeled Petri nets. *IEEE Transactions on Automation Science and Engineering 11*(1), 144–153.

Cassandras, C. G. and S. Lafortune (2008). *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc.

Cassez, F. and S. Tripakis (2008). Fault diagnosis with dynamic observers. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pp. 212–217.

Giua, A., C. Seatzu, and D. Corona (2007). Marking estimation of Petri nets with silent transitions. *IEEE Transactions on Automatic Control 52*(9), 1695–1699.

Hadjidj, R., H. Boucheneb, and D. Hadjidj (2007). Zenoness detection and timed model checking for real time systems. In *VECoS'07*, pp. 120–134.

Khansa, W., J. P. Denat, and S. Collart-Dutilleul (1996). P-time Petri nets for manufacturing systems. In *WODES'96, Edinburgh UK*, pp. 94–102.

Lin, F. (1994). Diagnosability of discrete event systems and its applications. *Discrete Event Dynamic Systems 4*(2), 197–212.

Merlin, P. and D. Faber (1976). Recoverability of communication protocols-implications of a theoretical study. *IEEE Trans. Comm. 24*(9), 381–404.

Murata, T. (1989). Petri nets, properties, analysis and applications. *Proceedings of the IEEE 77*, 541–580.

Sampath, M., R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis (1995). Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control 40*(9), 15551575.

Wang, X., C. Mahulea, and M. Silva (2013, 07/2013). Fault diagnosis graph of time petri nets. In *ECC'13: European Control Conference*, Zurich, Switzerland.

Wang, X., C. Mahulea, and M. Silva (2014). Model checking on fault diagnosis graph. In *12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14-16, 2014.*, pp. 434–439.

Zaytoon, J. and S. Lafortune (2013). Overview of fault diagnosis methods for discrete event systems. *Annual Reviews in Control 37*(2), 308 – 320.

Combining Enumerative and Symbolic Techniques for Diagnosis of Discrete-Event Systems

Abderraouf Boussif
Univ. Lille Nord de France,
F-59000 Lille, France
IFSTTAR, Cosys/Estas,
F-59650 Villenveuve d'Ascq, France
FR
abderraouf.boussif@ifsttar.fr

Mohamed Ghazel
Univ. Lille Nord de France,
F-59000 Lille, France
IFSTTAR, Cosys/Estas,
F-59650 Villenveuve d'Ascq, France
FR
mohamed.ghazel@ifsttar.fr

Kais Klai
LIPN, CNRS UMR 7030,
Univ. Paris 13, Sorbonne Paris Cité,
FR
kais.klai@lipn.univ-paris13.fr

In this paper, an efficient approach to verify diagnosability of discrete-event systems is proposed. The approach consists in constructing a hybrid diagnoser based on the symbolic observation graph (SOG), which is a technique that combines symbolic and enumerative representations in order to build a deterministic observer from a partially observed model. The construction of the diagnoser as well as the verification of diagnosability are performed simultaneously on-the-fly, which can considerably reduce the generated state space of the diagnoser and thus the overall running time. Furthermore, the proposed approach provides a heuristic strategy in order to converge fast into the necessary part, of the diagnoser, for analysing diagnosability.

Discrete-Event Systems, Diagnosability Analysis, Symbolic Observer Graph, On-the-Fly Verification.

1. INTRODUCTION

In automated monitoring and fault diagnosis of complex dynamic systems, one of the central tasks is to detect and identify the occurrence of failures as early as possible. This task has become an active research area in recent years (Zaytoon and Lafortune 2013). From the theoretical point of view and at a high level of abstraction, Discrete-Event Systems (DESs) are more suitable for performing diagnosis analysis on complex systems (Cassandras and Lafortune 2007).

One of the main issues in diagnosis activity that must be addressed is diagnosability investigation. Analysing diagnosability of a system intends to determine accurately whether any predetermined failure or class of failures can be detected and identified within a finite delay following its occurrence (Sampath et al. 1995).

Diagnosability verification has received considerable attention since the seminal paper by (Sampath et al. 1995), which provides a basic concept and a formal definition of diagnosability analysis and fault diagnosis of DESs that were adopted and further developed later. In this paper, (Sampath et al. 1995), the original definition of diagnosability was introduced in the language context. A systematic method to check diagnosability based on a dedicated deterministic version of the model derived from the original system, a so-called *diagnoser*, was also provided. It consists of a specific observer of the system associated with a labelling function that attributes to each state (or macro-state), in this observer, a label indicating whether the state is reached by a faulty execution or not, i.e. an execution where some particular unobservable events, called *faults*, have occurred or not.

Other automata-based approaches (Jiang et al. 2001; Yoo and Lafortune 2002), aiming to reduce computational complexity have been then proposed. In (Yoo and Lafortune 2002), a polynomial-time algorithm for checking diagnosability based on a structure called *verifier* is adopted. In (Jiang et al. 2001), an algorithm based on the twin plant structure (a parallel composition of the investigated automaton with itself) is proposed. Reformulations of these works in model-checking framework were first proposed in (Cimatti et al. 2003) and extended in (Boussif and Ghazel 2015). The goal is to check

diagnosability property in the same way as to check any safety property.

Furthermore, some works on diagnosability of DESs turned to Petri nets (PNs) formalism, benefiting from the mathematical and graphical representations capability and the well-developed theory underlying PNs (Peterson 1981). (Ushio et al. 1998) extended Sampath's study to systems modelled by PNs with the assumption that some places are observables whereas all of the transitions are unobservable. A diagnoser is constructed from the reachability graph. In (Wen and Li 2005), the authors proposed a sufficient condition for testing diagnosability by checking the structure of T-invariants of a PN. In (Cabasino et al. 2009) the modified basis reachability graph (MBRG) and basis reachability diagnoser (BRD), which provide a compact representation of the reachability graph, were developed. In (Basile et al. 2012), an approach for checking diagnosability by quantifying the finite delay of diagnosability (the so-called K-diagnosability) was proposed by using the integer linear programming (ILP) technique. A structure called verifier net (VN) was introduced in (Cabasino et al. 2014) to deal with diagnosability for both bounded and unbounded PNs. Recently, (Liu et al. 2014a) has proposed an on-the-fly and incremental diagnosis technique to construct a diagnoser from a bounded PN in order to verify diagnosability and K-diagnosability properties.

To get a general overview on the literature pertaining to diagnosis of DESs, the reader can refer to the recent survey in (Zaytoon and Lafortune 2013), where theoretical and practical issues, tools and other issues in relation with diagnosis are discussed.

The challenge of analysing diagnosability is the combinatorial explosion problem that appears during the building of the intermediate models (diagnoser, verifier, twin plant, MBRG, etc.). This is due to the high complexity of these constructed models and to the generation of the whole state-space which may have considerable time and memory cost.

To partially overcome this problem, we propose, in this paper, an efficient approach to construct a hybrid diagnoser on-the-fly, in the sense of combining enumerative and symbolic techniques. The contributions of this paper are twofold:

1. We provide a behavioural diagnoser based on the Symbolic Observation Graph (Haddad et al. 2004) which is an efficient binary decision diagram (BDD) based abstraction of the model state space. Thus, macro-states of the diagnoser will be compacted using BDDs while transitions between macro-states are represented by enumerate observable events.
2. We design an appropriate algorithm, for simultaneously constructing the diagnoser and checking diagnosability on-the-fly. Actually, the verification process is stopped (only a part of the diagnoser is built) as soon as the diagnosability is proven to be unsatisfied, which can considerably reduce the generated state space of the diagnoser. Furthermore, the proposed algorithm is endowed with a heuristic strategy in order to converge fast into the necessary part of the diagnoser for diagnosability analysis.

The paper is structured as follows. In Section 2, we introduce the basic background needed to deal with diagnosability and to develop our approach. In Section 3, we recall the notion of diagnosability as well as the original diagnoser approach. Section 4 is devoted to discuss the Symbolic Observation Graph adapted to the context of this paper. In Section 5, the verification approach is sketched out then an on-the-fly algorithm based on the SOG is presented. Section 6 discusses the pertinent existing work in relation with the present work. Finally, conclusion remarks and future research directions are given in Section 7.

2. PRELIMINARIES

We first recall some standard notations that will be used in the sequel. Let Σ be a finite alphabet of events (actions). A string is a finite sequence of events in Σ. ϵ denotes the empty string. Given a string s, the length of s is denoted by $|s|$. The set of all strings formed by events in Σ is denoted by Σ^*. Any subset of Σ^* is called a *language*. Given a string $s \in L$, $L/s \triangleq \{t \in \Sigma^* | s.t \in L\}$ is called the *post-language* of L after s and defined as L/s. L is said to be *extension-closed* when $L.\Sigma^* = L$.

The approach introduced in this paper applies to discrete-events systems modelled by Labelled Transitions Systems (LTSs for short). The formal definition of LTS is as follows.

Definition 1 *(LTS): An LTS over Σ is defined by a 4-tuple $\langle Q, \Sigma, \rightarrow, q_0 \rangle$, where:*

- *Q is a finite set of states;*
- *Σ is a finite set of events;*
- *$\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation;*
- *$q_0 \in Q$ is the initial state.*

In the remainder of this section, we consider a given LTS $G = \langle Q, \Sigma, \rightarrow, q_0 \rangle$. For $q, q' \in Q$ and $\sigma \in \Sigma$, we denote $q \xrightarrow{\sigma} q' \triangleq (q, \sigma, q') \in \rightarrow$. $q \rightarrow$ means that $\exists q' \in Q : q \xrightarrow{\sigma} q'$. If $s = \sigma_1, \sigma_2, \ldots, \sigma_n$ is a string (sequence of events), $\bar{s}$ denotes the set of actions occurring in s. Moreover, $q \xrightarrow{s} q'$ denotes that $\exists q_1, q_2, \ldots, q_{n-1} \in Q$ such that, $q \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \ldots q_{n-1} \xrightarrow{\sigma_n} q'$. $q \xrightarrow{*} q'$ denotes that q' is reachable from q (i.e. $q \xrightarrow{s} q'$ for some $s \in \Sigma^*$), and $q \xrightarrow{*}_E q'$ holds if $\bar{s} \subseteq E$.

We denote by $Enable(q)$ the set of events σ s.t. $q \xrightarrow{\sigma}$, for a set of states $Q' \subseteq Q$, $Enable_E(Q')$ denote the set of enabled events from the set of states Q', i.e. $Enable(Q')$ denotes $\bigcup_{q \in Q'} Enable(q)$.

An execution from the initial state q_0 of an LTS G is a finite sequence of transitions $\pi = q_0 \xrightarrow{\sigma_1} q_1 \ldots \xrightarrow{\sigma_n} q_n$. The event-trace of π, denoted by $Tr(\pi)$, is the sequence of events $\sigma_1, \ldots, \sigma_n$, $\pi[i]$ stands for the prefix of π truncated at state q_i, i.e., $\pi[i] = q_0 \xrightarrow{\sigma_1} q_1 \ldots \xrightarrow{\sigma_i} q_i$ and $last(\pi)$ represents the last state of π. The set of finite executions of LTS G from the initial state q_0 is denoted by $Runs(G)$. The behaviour of G is described by its language $L(G) = \{s \in \Sigma^* | q_0 \xrightarrow{s}\}$.

As we are interested in diagnosis issues, partial observation plays a central role. In this regard, some events in Σ are observable, i.e. their occurrence can be observed, while the rest are unobservable. Thus, the event set Σ can be partitioned as $\Sigma = \Sigma_o \biguplus \Sigma_u$, where Σ_o denotes the set of observable events and Σ_u the set of unobservable events. To reflect the limitation on observation, we define the projection function $P : \Sigma^* \rightarrow \Sigma_o^*$. In the usual manner, $P(\sigma) = \sigma$ for $\sigma \in \Sigma_o$; $P(\sigma) = \epsilon$ for $\sigma \in \Sigma_u$, and $P(s\sigma) = P(s)P(\sigma)$, where $s \in \Sigma^*$, $\sigma \in \Sigma$. Thus, P simply erases the unobservable events in any event-trace. The inverse projection operation P_L^{-1} is defined by $P_L^{-1}(y) = \{s \in L(G) : P(s) = y\}$. Any two executions π and π' are called *indistinguishable* with respect to the projection function P if they can generate the same observed event-trace. With a slight abuse of notation, we write $P(\pi) = P(\pi')$ if π and π' are *indistinguishable*.

In the context of fault diagnosis, let $\Sigma_f \subseteq \Sigma_u$ denote the set of failure events. They are usually represented using unobservable events, since their detection and diagnosis would be trivial if they were observable. We partition the set of failure events into disjoint failure classes $\Sigma_f = \Sigma_{f_1} \biguplus \Sigma_{f_2} \biguplus \cdots \biguplus \Sigma_{f_m}$, with Σ_{f_i} denotes the failure class f_i.

3. DIAGNOSABILITY ANALYSIS

In this work, only diagnosability analysis of permanent faults is considered. Once a fault has occurred, the system remains irreparably faulty. We assume that the LTS G under consideration satisfies the following two assumptions:

1. The language generated by G is live, i.e. there is an executable transition from any state of the system.
2. The LTS G is finite, in term of the state space, and does not contain cycles formed only by unobservable events.

3.1. Definition of diagnosability

Diagnosability is an important property in the monitoring and fault diagnosis activities. In simple terms, it refers to the ability to infer accurately, from a partially observed execution, about the faulty behaviour within a finite delay after possible occurrences of faults. The original definition of diagnosability, introduced in the seminal work of (Sampath et al. 1995) is recalled in the following.

Definition 2 *(diagnosability (Sampath et al. 1995)) A prefix-closed and live language L is said to be diagnosable with respect to the projection function P and with respect to a failure class of faults Σ_f if the following holds*

$$(\exists n_i \in \mathbb{N})\ [\forall s \in \Psi(\Sigma_f)]\ (\forall t \in L/s)\ [|t| \geq n_i \Rightarrow D]$$

where the diagnosability condition D is

$$\omega \in P^{-1}[P(s.t)] \Rightarrow \Sigma_f \in \omega.$$

with $\Psi(\Sigma_f)$ is the set of finite sequences that terminate with a faulty event from Σ_f.

The above definition means the following. Let s be any sequence generated by the LTS G that ends with a failure event from Σ_f, and let t be any sufficiently long continuation of s. Condition D then requires that every sequence ω belonging to the language that produces the same observable trace as sequence $s.t$ ($P(\omega) = P(s.t)$) must hold a failure event from Σ_f. In other terms, diagnosability requires that every failure event leads to observations distinct enough to identify the failure type within a finite delay.

3.2. Verification of diagnosability

In order to analyse diagnosability, (Sampath et al. 1995) has proposed a systematic approach that consists in building a specific model called *diagnoser*. It is a deterministic automaton whose transitions correspond to the observable events of the system and whose states are estimation system state associated with labels to indicate if a state is reached by an observable trace containing a faulty event or not.

Definition 3 *(Diagnoser (Sampath et al. 1995))*

Let $G = \langle Q, \Sigma, \rightarrow, q_0 \rangle$ be an LTS to be diagnosed. A diagnoser of G is a deterministic LTS $G_d = \langle \mathcal{X}, \Sigma_o, \rightarrow_d, \mathcal{X}_0 \rangle$ associated with a tagging function $Diag : \mathcal{X} \rightarrow 2^{\triangle}$, with $\triangle = \{N, F\}$ (for only one class of failures). with N means normal and F means faulty.

Each diagnoser state x has the form $x = \{(q_1, l_1), \ldots, (q_n, l_n)\}$, with $q_i \in Q$ and $l_i \in \triangle$. If $\forall i = 1, \ldots, n$, we have $l_i = N$ (resp. $l_i = F$), the diagnoser state x is said to be $N\text{-}certain$ (resp. $F\text{-}certain$), otherwise $F\text{-}uncertain$ state.

For more details about the formal framework and algorithmic procedure of constructing the diagnoser, we refer the reader to the original paper (Sampath et al. 1995).

We define an f-indeterminate cycle in a diagnoser to be a cycle composed exclusively of $F\text{-}uncertain$ diagnoser states and corresponding to the presence of two cycling traces, in the system, that sharing the same observable events, such that the faulty event f from the class Σ_f occurs in the 1st trace but not in the 2nd. The notion of f-indeterminate cycle is very important, since it helps to give a necessary and sufficient condition for diagnosability analysis.

Theorem 1 *((Sampath et al. 1995))*

A system modelled by an LTS G is diagnosable if and only if there are no f-indeterminate cycles in its diagnoser G_d for any class of faults Σ_f.

3.3. Example

Let us consider the LTS G in Figure 1 (adapted from (Sampath et al. 1995)). The set of observable events is $\Sigma_o = \{a, b, d, t\}$ and the set of unobservable events is $\Sigma_u = \{u, f\}$ with f a faulty event in Σ_f.

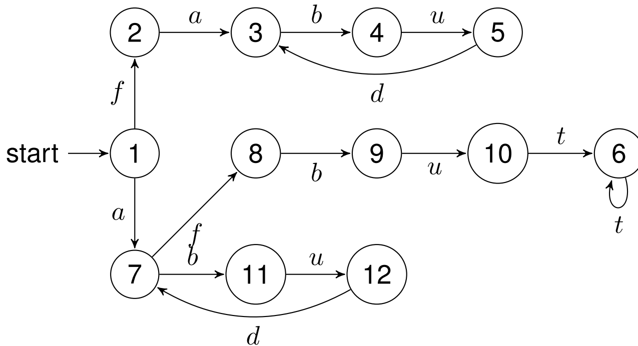


Figure 1: The LTS G

The diagnoser G_d corresponding to the LTS G is depicted in Figure 2. There exists a cycle of f-uncertain states composed of $\{3F, 7N\}$ and $\{5F, 10F, 12N\}$ w.r.t. the observable sequence $(bd)^*$. This cycle corresponds to two cycles in the LTS G. The 1st one, which is composed of states $\{3\}, \{4\}, \{5\}$ w.r.t. the sequence $(bud)^*$, which is reachable from the faulty sequence fa. The 2nd cycle, which is composed of states $\{7\}, \{11\}, \{12\}$ and reached by the fault-free sequence a. Thus we can infer, according to Theorem 1, that there exists an f-indeterminate cycle in the diagnoser and consequently the LTS G is not diagnosable w.r.t. to faulty class Σ_f and the projection function P.

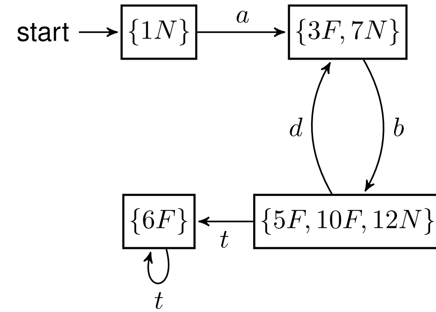


Figure 2: Diagnoser G_d of the LTS G

It is worth noticing that the diagnoser can be used either off-line to check diagnosability or on-the-fly by connecting it to the system in order to provide on-line diagnosis upon the occurrence of observable events.

4. SYMBOLIC OBSERVATION GRAPH (SOG)

In this section, we present the so-called *symbolic observation graph* (Haddad et al. 2004) and we show how it is used to abstract LTS behaviour. In (Haddad et al. 2004), the authors have introduced the SOG as an abstraction of the reachability graph of concurrent systems and showed that the verification of an event-based formula of LTL/X on the SOG is equivalent to the verification on the original reachability graph. The construction of the SOG is guided by the set of observable events. The SOG is defined as a graph where each node is a set of states linked by unobserved events and each arc is labelled with an observable event. The SOG nodes are called *aggregates* and may be represented and handled efficiently using decision diagram techniques (BDDs, see for instance (Bryant 1992)). The SOG is said to be hybrid since it is both an explicit and a symbolic structure: the graph is represented explicitly while the nodes are sets of states encoded and managed symbolically. Despite the exponential theoretical complexity of the size of a SOG, it has a very moderate size in practice (see

(Haddad et al. 2004; Klai and Petrucci 2008); (Klai and Poitrenaud 2008) for experimental results).

In the following, we first define what an aggregate is formally, before providing a formal definition of a SOG associated with an LTS.

Definition 4 *(aggregate)*

Consider the LTS $G = \langle Q, \Sigma, \rightarrow, q_0 \rangle$ *with* $\Sigma = \Sigma_o \biguplus \Sigma_u$. *An aggregate* a *is a non empty set of states satisfying:* $q \in a \Leftrightarrow Saturate_{\Sigma_u}(q) \subseteq a$; *where* $Saturate_{\Sigma_u}(q) = \{q' \in Q | q \xrightarrow{*}_{\Sigma_u} q'\}$.

In the following, $Saturate_{\Sigma_u}$ is extended to sets of states as follows: $Saturate_{\Sigma_u}(Q') = \bigcup_{q \in Q'} Saturate_{\Sigma_u}(q)$.

Definition 5 *(symbolic observation graph)*

The deterministic symbolic observation graph SOG(G*) associated with an LTS* G *is an LTS* $\langle \mathcal{A}, \Sigma_o, \rightarrow_{\Sigma_o}, a_0 \rangle$ *where:*

1. $\mathcal{A}$ *is a finite set of aggregates such that:*
   a) *There is an aggregate* $a_0 \in \mathcal{A}$ *s.t.* $a_0 = Saturate_{\Sigma_u}(q_0)$;
   b) *For each* $a \in \mathcal{A}$ *and for each* $\sigma \in \Sigma_o$, *if* $\exists q \in a, q' \notin a : q \xrightarrow{\sigma} q'$ *then* $Saturate(\{q' \notin a | \exists q \in a, q \xrightarrow{\sigma} q'\})$ *equals* a' *for some aggregate* a' *and* $(a, \sigma, a') \in \rightarrow_{\Sigma_o}$;
2. $\rightarrow_{\Sigma_o} \subseteq \mathcal{A} \times \Sigma_o \times \mathcal{A}$ *is the transition relation, obtained by applying* $1.b$).

The SOG can be constructed by starting with the initial aggregate a_0 and iteratively adding new aggregates as long as the condition of $1.b$) holds (see (Haddad et al. 2004) for a construction algorithm).

5. USING SOGS TO ANALYSE DIAGNOSABILITY

In this section, we discuss how the SOG is used in order to build a hybrid diagnoser and we provide an on-the-fly algorithm to construct the hybrid diagnoser and to verify diagnosability simultaneously.

The underlying idea behind using SOG for diagnosability analysis is basically to tackle the state explosion phenomenon raised when building the diagnoser. It is worth recalling here that the Sampath's diagnoser is computed with an exponential complexity regarding the state space of the model (Sampath et al. 1995). Obviously, this represents a serious limit of the diagnoser based approach when large models are handled.

Using the results of Section 4, we would use the SOG to perform a hybrid diagnoser construction. In order to capture the feature of analysing diagnosability which is tracking the ambiguous behaviour of the system, i.e. normal and faulty executions which share the same observable events, we modify the structure of the aggregate, introduced in Definition 4, by splitting the set of states (managed using BDDs) into tow sets of states in the same aggregate: one set contains normal states, i.e. states reachable by fault-free sequences, and the other contains faulty states, i.e. states reachable by sequences containing one (or more) faulty events. Both of sets are represented and managed using BDDs.

Figure 3 depicts the general form of a diagnoser aggregate where two BDDs represent two sets of states; BDD_n represents the set of normal states, while BDD_f represents the set of faulty ones. The set of faulty states may be reached from the set of normal states by the occurrence of a faulty event, and thus a faulty transition form BDD_n to BDD_f may exist in any diagnoser aggregate. Depending on the executed behaviour, i.e. the executed sequence, the diagnoser aggregate may contain the two sets of states (BDDs) or only one set. If a diagnoser aggregate contains only BDD_n (resp. BDD_f), it is called an N-$certain$ (resp. F-$certain$) diagnoser aggregate, else it is an F-$uncertain$ diagnoser aggregate in the same way as in the classic diagnoser (Definition 3).

The dashed arrows show the different possibilities that a transition from a diagnoser aggregate can produce. For instance, observable event b enabled by the diagnoser aggregate can be enabled from both normal and faulty sets or from only one set. This feature will be considered during the on-the-fly construction of our hybrid diagnoser, since we do not need to construct the diagnoser aggregates reached through an observable event from only the faulty set of an aggregate. Moreover, this information will be used to establish some heuristics that prioritizing the branches to be followed while building the hybrid diagnoser.

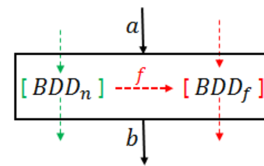


Figure 3: A diagnoser aggregate

Definition 6 *(diagnoser aggregate)*

Consider an LTS $G = \langle Q, \Sigma, \rightarrow, q_0\rangle$ with $\Sigma = \Sigma_o \uplus \Sigma_u$ and $\Sigma_f \subset \Sigma_u$. A diagnoser aggregate $a = \langle Q_n, Q_f\rangle$ is a non empty set of states satisfying:

1. *$\forall q \in Q$ s.t. $q_0 \xrightarrow{*f*} q$ (i.e. q is reachable by a faulty sequence): $q \in a \Leftrightarrow Saturate_{\Sigma_u}(q) \subseteq a.Q_f$;*
2. *$\forall q \in Q$ s.t. $q_0 \xrightarrow{*}_{\Sigma\setminus\Sigma_f} q$ (i.e. q is reachable by a fault-free sequence): $q \in a \Leftrightarrow Q' = Saturate_{\Sigma_u\setminus\Sigma_f}(q) \subseteq a.Q_n \wedge Saturate_{\Sigma_u}(Img(Q', f) \subseteq a.Q_f$.*
3. *$\forall q, q' \in a, \exists s, s' \in \Sigma^*$, s.t. $q_0 \xrightarrow{s} q$, $q_0 \xrightarrow{s'} q'$, and $P(s) = P(s')$*

with $Saturate_{\Sigma_u\setminus\Sigma_f}(q)$=$\{q' \in Q | q \rightarrow_{\Sigma_u\setminus\Sigma_f} q'\}$, and $Img(Q', f)$=$\{q' | q \in Q' : \quad q \xrightarrow{f} q'\}$, i.e. it returns the set of immediate successors of states in Q' through the occurrence of event f.

To simplify the notation, we denote by $a.Q_n$ (resp. $a.Q_f$) the set of normal (resp. faulty) states in an aggregate a.

5.1. Constructing the hybrid diagnoser

We now introduce the hybrid diagnoser which is a modified SOG built from the LTS model G.

Definition 7 *(hybrid diagnoser)*

The hybrid diagnoser $D_{SOG}(G)$ associated with an LTS G is a modified SOG $\langle \Gamma, \Sigma_o, \rightarrow_{SOG}, \Gamma_0\rangle$.

1. *Γ is a finite set of diagnoser aggregates.*
2. *Γ_0 is the initial diagnoser aggregate with;*
   a) *$\Gamma_0.Q_n$ = $Saturate_{\Sigma_u\setminus\Sigma_f}(q_0)$;*
   b) *$\Gamma_0.Q_f$ =$Saturate_{\Sigma_u}(Img(\Gamma_0.Q_n, f))$.*
3. *$\rightarrow_{SOG} \subseteq \Gamma \times \Sigma_o \times \Gamma$ is the transition relation, defined as below,*
   $\forall a, a' \in \Gamma, \sigma \in \Sigma_o$ s.t. $\sigma \in Enable(a.Q_n \cup a.Q_f)$:
   $a \xrightarrow{\sigma}_{SOG} a' \Leftrightarrow a'.Q_n$= $Saturate_{\Sigma_u\setminus\Sigma_f}(Img(a.Q_n, \sigma))$ $\wedge$ $a'.Q_f$=$Saturate_{\Sigma_u}(Img(a'.Q_n, f) \cup Img(a.Qf, \sigma))$

To summarize, the hybrid diagnoser $D_{SOG}(G)$ is constructed as follows: let the current aggregate of the diagnoser be a, and the next observed event be σ. The target diagnoser aggregate a' of the hybrid diagnoser is computed following these rules:

1. If $\sigma \in Enable(a.Q_n) \cap Enable(a.Q_f)$ then:
   a. $a'.Q_n = Saturate_{\Sigma_u\setminus\Sigma_f}(Img(a.Q_n, \sigma))$.
   b. $a'.Q_f = Saturate_{\Sigma_u}(Img(a'.Q_n, f) \cup Img(a.Qf, \sigma))$.
2. If $\sigma \in Enable(a.Q_n)\setminus Enable(a.Q_f)$ then:
   a. $a'.Q_n = Saturate_{\Sigma_u\setminus\Sigma_f}(Img(a.Q_n, \sigma))$.
   b. $a'.Q_f = Saturate_{\Sigma_u}(Img(a'.Q_n, f))$.
3. If $\sigma \in Enable(a.Q_f)\setminus Enable(a.Q_n)$ then:
   a. $a'.Q_n = \emptyset$.
   b. $a'.Q_f = Saturate_{\Sigma_u}(Img(a.Q_f, \sigma))$.

These rules preserve a specific fault propagation. From an F-$uncertain$ diagnoser aggregate, we can reach either another F-$uncertain$, an F-$normal$ or an F-$certain$ diagnoser aggregate, from an N-$certain$ diagnoser aggregate, we can reach either another N-$certain$ diagnoser aggregate or an F-$uncertain$ one, and finally from an F-$certian$ diagnoser aggregate, we can reach only another F-$certain$ diagnoser aggregate, which depicts exactly the hypothesis of permanent failures. Figure 4 illustrates these points.

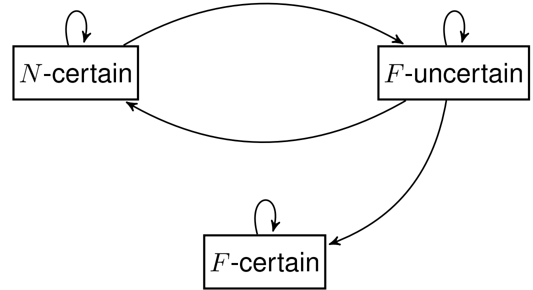


Figure 4: Fault propagation on the hybrid diagnoser

Example 1 *Figure 5 represents the hybrid diagnoser associated with the LTS of Example 1, depicted in Figure 1. As it is intended to construct the hybrid diagnoser on-the-fly, it is more convenient to represent the hybrid diagnoser as a tree-like structure.*

The initial aggregate composed of the initial state of the LTS and the state 2 reachable from state 1 by the occurrence of faulty event f. Both of the diagnoser aggregates (2) and (3) contain two sets of states (each one is represented by a BDD). After the occurrence of event d we reach diagnoser aggregate (4), which is the same as diagnoser aggregate (2) thus, there exists a cycle on the hybrid diagnoser composed of aggregates (2) and (3) by executing the observable event sequence $(bd)^*$. Diagnoser aggregate (5) is reached after the occurrence of event t and it contains only the set of faulty states thus, it is an F-$certain$ diagnoser aggregate. As F-$certain$ diagnoser aggregates are

not necessary to analyse diagnosability, in on-the-fly constructing of the hybrid diagnoser, we do not construct them. Indeed, one knows that all the subsequent aggregates will be F-$certain$ as well. Besides, computing such aggregates is not necessary for online diagnosis.

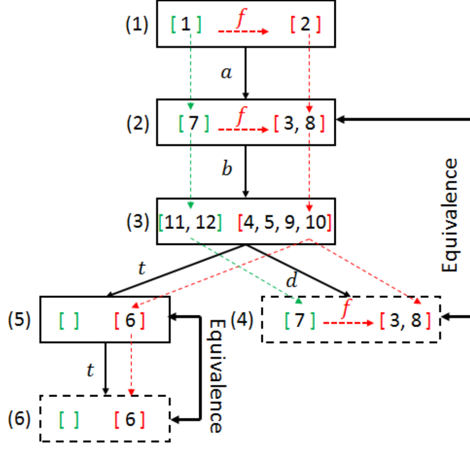


Figure 5: Hybrid diagnoser of the LTS in Figure 1.

We recall that our goal is to avoid the state-explosion problem, not only by providing this compact form (SOGs) to build the hybrid diagnoser but also by constructing the hybrid diagnoser on-the-fly and verifying diagnosability simultaneously. Constructing the hybrid diagnoser on-the-fly serves to avoid generating the whole state space of the diagnoser even if the system is diagnosable, i.e. as we deal with permanent faults, we do not need to construct the part, of the hybrid diagnoser, containing only F-certain diagnoser aggregates since such a part is not necessary for analysing diagnosability (see (Liu et al. 2014a) for more details).

Hereafter, we provide the SOG-based algorithm needed for on-the-fly construction of the hybrid diagnoser. The following function and data structures are used:

- $Img(S, t)$, as described previously, returns the set of immediate successors of the states of a set S through the occurrence of event t.
- OBDDs (Ordered Binary Decision Diagram) are used to represent the sets of states belonging to an aggregate, i.e. the set of normal states and the set of faulty states in an aggregate. This task is performed by the function Reduce().
- The hybrid diagnoser is represented by a standard graph representation with a set of vertices, namely V, and a set of edges, namely E, connecting these aggregates and labelled with observable events.
- $EnableObs(S)$ returns the set of observed events that are enabled by at least one of the states in set S.
- $Saturate_{\Sigma_i}()$, as defined before, computes the various states reached through events from set Σ_i.
- $Stack$ is an ordered set of 5-uplet, which contains two sets of states (S_n, S_f) and three sets of events (Evt_f, Evt_n, Evt) with $Evt = (Evt_n \cup Evt_f) \backslash (Evt_n \cap Evt_f)$.
- $IsUncertain()$ is a function that returns Boolean value ($true$ if the encountered cycle is composed of only f-$uncertain$ diagnoser aggregates and $false$ otherwise).
- $IsIndeterminate()$: is a function that returns Boolean value ($true$ if the existing cycle is an f-indeterminate cycle and $faulse$ otherwise. This function will be discussed later.
- $RemoveLast(S)$ is an operation that removes, then returns, the last event of a set S.
- For the sake of simplicity, we consider that Σ_f contains only one faulty event, generalization to a set of faulty events is straightforward.

The initialization step (lines 1-11) serves to compute the initial diagnoser aggregate, handle it efficiently using OBDD (function Reduce()), and push it, associated with its enabled observable events, into the stack. The construction of the hybrid diagnoser is performed using a depth first exploration: As long as the stack in not empty, a new observable event t, enabled by the diagnoser aggregate a at the top of the stack, is removed from the set of enabled events (Evt) and then processed. If such an event does not exist, the corresponding aggregate is poped from the stack (lines 13-15). Otherwise, if the set Evt_n of events enabled by the set of normal states S_n of the diagnoser aggregate a, is empty then the aggregate a is poped from the stack (lines 16 and 18). This step serves to avoid the construction of the subsequent F-$certain$ diagnoser aggregates, i.e. since this part of the hybrid diagnoser is not necessary to analyse diagnosability.

The computation of the new aggregate a', reachable through an observable event from aggregate a, is completed by saturation on the unobservable events (lines 20-26). If a' has already been encountered (i.e. existence of a cycle) then the hybrid diagnoser is updated by adding a new edge (lines 27-28) and if the cycle is $uncertain$ (i.e., contains only f-$uncertain$ diagnoser aggregates), the function $IsIndeterminate()$ is launched in order to detect whether there exists an f-indeterminate cycle or not (line 29-32). If the cycle is an f-indeterminate

one then we output that the model is undiagnosable and we stop the diagnoser construction. Otherwise, construction is continued, a' with its enabled observable events are pushed into the stack, and so on. When the stack is empty, then the necessary part of the diagnoser, for analysing diagnosability, is completely built, we output that the model is diagnosable.

As mentioned in Section 3, diagnosability analysis is performed by searching two infinite executions that share the same observed event-sequences such that one sequence contains a faulty event and not the other one. That means to search an f-indeterminate cycle in the diagnoser (Sampath et al. 1995). The same procedure is used in our case, i.e. searching f-indeterminate cycles in the hybrid diagnoser. Two steps are needed to check the existence of f-indeterminate cycles when a cycle of $F\text{-}uncertain$ diagnoser aggregate is found in the hybrid diagnoser:

1. Extract the observed event-sequence that leads to this cycle (of $F\text{-}uncertain$ diagnoser aggregates).
2. Check if this observed event-sequence corresponds to two cycle in the LTS model. One cycle is reached by a fault-free event-sequence and the other one is reached by a faulty event-sequence.

This task is performed by function $IsIndeterminate()$ in the Algorithm 1 (line 28) which calls a specific function ($path_exists()$) from the $digraph$ library (Rushton 2012). ($digraph$ is a library dedicated for searching cycles from the system model).

We emphasize that verification of the existence of f-indeterminate cycles is performed on-the-fly in parallel to the process of constructing the hybrid diagnoser, i.e. the process of constructing the hybrid diagnoser runs and when a cycle of $F\text{-}uncertain$ diagnoser aggregates is found, we check whether this cycle corresponds to an f-indeterminate cycle or not. If it is the case (i.e. the cycle is an f-indeterminate cycle), then the whole process is stopped and a negative verdict is emitted regarding diagnosability, else the building process is continued.

Example 2 *Let us take again LTS G of Figure 1 and its hybrid diagnoser (Figure 5). We have a cycle, in the hybrid diagnoser, composed of only $F\text{-}uncertain$ diagnoser aggregates $(2) \rightleftharpoons (3)$. Once the algorithm of construction arrives at this cycle, we check if this cycle is an f-indeterminate one or not, before continuing the construction process. The cycle of $f\text{-}uncertain$ diagnoser aggregates $(2) \rightleftharpoons (3)$ is reached by executing the observed event-sequence $a(bd)^*$.*

Algorithm 1 On-the-fly algorithm to construct the hybrid diagnoser

```
DiagSOG (LTS, Σ_o, Σ_u, f);
Diagnoser aggregate a, a';
Set of vertices V;
Set of edges E;
Set of Events Evt_n, Evt_f, Evt;
Set of states S_n, S_f, S'_n, S'_f;
Stack st = ∅;

 1: S_n = Saturate_{Σ_u\f}(q_0);
 2: S_f = Img(S_n, f);
 3: if (S_f ≠ ∅) then
 4:     S_f = Saturate_{Σ_u}(S_f);
 5: end if
 6: a = ⟨S_n, S_f⟩;
 7: Reduce (a, Σ_u);
 8: V = a; E = ∅;
 9: Evt_n = EnableObs(S_n);
10: Evt_f = EnableObs(S_f);
11: st.Push(⟨S_n, S_f, Evt_n, Evt_f, Evt⟩;
12: while (st ≠ ∅) do
13:     ⟨S_n, S_f, Evt_n, Evt_f, Evt⟩ = st.Top();
14:     if Evt = ∅ then
15:         ⟨S'_n, S'_f, Evt_n, Evt_f, Evt⟩ = st.Pop();
16:     else
17:         if (Evt_n = ∅ then
18:             ⟨S'_n, S'_f, Evt_n, Evt_f, Evt⟩ = st.Pop();
19:         else
20:             t = RemoveLast(Evt);
21:             S'_n = Img(S_n, t);
22:             S'_n = Saturate_{Σ_u\f}(S'_n);
23:             S'_f = Img(S_f, t) ∪ Img(S'_n, f);
24:             S'_n = Saturate_{Σ_u}(S'_f);
25:             a' = ⟨S'_n, S'_f⟩;
26:             Reduce (a', Σ_u);
27:             if (∃w ∈ V | w = a'); then
28:                 E = E ∪ a -t->_SOG w;
29:                 if (IsUncertain()) then
30:                     if (IsIndeterminate()) then
31:                         return UNDIAGNOSABLE;
32:                     end if
33:                 end if
34:             else
35:                 V = V ∪ {a'};
36:                 E = E ∪ a -t->_SOG a';
37:                 Evt_n = EnableObs(S'_n);
38:                 Evt_f = EnableObs(S'_f;
39:                 st.Push⟨S'_n, S'_f, Evt_n, Evt_f, Evt⟩;
40:             end if
41:         end if
42:     end if
43: end while
44: return DIAGNOSABLE;
```

In LTS G, the observed event-sequence $a(bd)^*$ corresponds to two event-sequences:

1. The faulty sequence $fa(bud)^*$ that leads to a cycle composed of states 3, 4, and 5.
2. The fault-free sequence $a(bud)^*$ that leads to a cycle composed of states 7, 11, 12.

Thus, we can infer that the cycle, in the hybrid diagnoser, composed of diagnoser aggregates (2) and (3) is an f-indeterminate cycle. Thus, we stop constructing the hybrid diagnoser and we output that LTS G is non diagnosable with respect the fault f.

5.2. A heuristic strategy to improve the building algorithm

Our algorithm for constructing the hybrid diagnoser is based on a depth-first search (DFS) to investigate the state space (diagnoser aggregate in the developed tree-like structures) execution by execution. Generally, no rules are defined to select the execution to be investigated first, i.e. the order of execution exploring is arbitrary. However, as we deal with diagnosability analysis, in our case, the diagnoser aggregate structure provides some information that can be exploited to direct the search in such a way as to increase the chances of quickly obtaining a diagnosability verdict by exploring the most promising executions at first.

When we deal with diagnosability analysis, the interesting executions of the system are those which share the same observed event-sequence such that some of them contain a faulty event and the others are fault-free. This is reduced to track the observed event-sequences, in the hybrid diagnoser, leading to $F\text{-}uncertain$ aggregates. Generally, there exists three types of enabled transitions from any aggregate, as depicted in Figure 6.

1. Transitions enabled only by states from the faulty set (Figure 6.(a)). As said before, this type of branches will not be explored.
2. Transitions enabled only by states from the normal set (Figure 6.(b)). In this case, we need to continue the construction since other faults may occur in the future.
3. Transitions enabled from both normal and faulty sets (Figure 6.(c)). In this case, the reached diagnoser aggregate will be certainly $F\text{-}uncertain$.

This last type of transitions is the most-promising to find an f-indeterminate cycle since we know, a priori, that the new diagnoser aggregate will be certainly an $f\text{-}uncertain$ aggregate, contrary to the other above cases. Thus, it will be the first to be explored in order

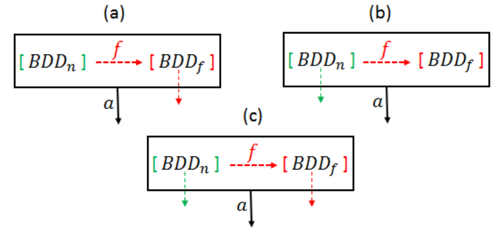


Figure 6: Types of enabled transitions from an aggregate

to direct the construction of the hybrid diagnoser and to potentially speed up the verification process. We note that in the actual version of the algorithm, this heuristic strategy is not implemented.

6. RELATED WORKS

In the literature, there are several diagnoser-based approaches for analysing diagnosability inspired from the seminal work of (Sampath et al. 1995). In (Zad et al. 2003) a state-based approach for on-line passive fault diagnosis was introduced. In the state-based approach, it is assumed that the set of states of the system model can be partitioned according to the faulty or normal condition of the system. In this work, a specific diagnoser is constructed as a finite state machine that takes information from the system (i.e. sequences of inputs/outputs) and generates an estimate of the condition of the system (i.e., faulty or normal). Establishing of this diagnoser has exponential time complexity. However, a model reduction scheme with polynomial time complexity is proposed to reduce the computational complexity of the procedure.

(Schumann et al. 2004) propose a symbolic framework based on binary decision diagrams for the diagnosis of DESs. A symbolic version of Sampath's diagnoser was proposed, while requiring considerably lower space and time than the enumerative approach of (Sampath et al. 1995). Recently, (Liu et al. 2014a) propose an on-the-fly algorithm for constructing and checking diagnosability of discrete-event systems modelled by LPNs using an enumerative approach. The goal is to avoid the construction of the whole state-space of the diagnoser especially when the system is not diagnosable. The approach was experimented over a Petri net benchmark and the obtained results were promising compared to those of Sampath's approach. A tool, called OF-PENDA (Liu et al. 2014b), was developed based on this approach to analyse diagnosability, K-diagnosability and K_{min}-diagnosability of systems, modelled by Labelled Petri Nets.

The approach proposed in this paper, takes advantage from these two last approaches (i.e., (Schumann et al. 2004) and (Liu et al. 2014a)) by combining the symbolic representation of diagnoser states and on-the-fly techniques for constructing the hybrid diagnoser and verifying diagnosability. We believe that this approach will improve efficiently analysis of diagnosability in terms of runtime and memory resources. We still need to apply the approach on several benchmarks in order to assess its efficiency. Indeed, determining analytical complexity while considering the worst case is not appropriate for such an on-the-fly approach.

7. CONCLUSION

In this work, we have developed an on-the-fly approach for diagnosability analysis, based on a hybrid diagnoser. The approach is based on the symbolic observation graph (SOG), which is a paradigm that combines symbolic and enumerative representations in order to build a deterministic observer from a partially observed model. The approach aims to improve the efficiently in terms of runtime and memory resources when analysing diagnosability.

Several future directions are considered. First, we wish to make experimentations over case-studies in order to assess the efficiency and the scalability of our approach and also to compare the obtained results with those provided by other existing approaches. Then, we will investigate some other practical versions of diagnosability, namely K-diagnosability and K_{min}-diagnosability. Finally, we intend to extend the proposed approach for analysing diagnosability based on the verifier approach by means of a non deterministic version of the symbolic observation graph.

REFERENCES

J. Zaytoon and S. Lafortune. Overview of fault diagnosis methods for discrete event systems. *Annual Reviews in Control*, pages 308–320, 2013.

C. G. Cassandras and S. Lafortune. Introduction to discrete event systems. *Second Edition, Springer*, 2007.

M. Sampath, R. Sengupta, and S. Lafortune. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, pages 1555–1575, 40(9), 1995.

S. Jiang, Z. Huang, V. Chandra, and R. Kumar. A polynomial algorithm for testing diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, pages 46(8), 1318–1321, 2001.

T. S. Yoo and S. Lafortune. Polynomial-time verification of diagnosability of partially observed discrete-event systems. *IEEE Transactions on Automatic Control*, pages 47(9), 1491–1495, 2002.

A. Cimatti, C. Pecheur, and R. Cavada. Formal verification of diagnosability via symbolic model checking. *Int. Joint Conference on Artificial Intelligence*, 2003.

A. Boussif and M. Ghazel. Diagnosability analysis of input/output discrete event system using model checking. *The 5th International Workshop on Dependable Control of Discrete Systems*, 2015.

J. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, 1981.

T. Ushio, I. Onishi, and K. Okuda. Fault detection based on Petri net models with faulty behaviors. *Systems, Man, and Cybernetics*, pages 113–118, 1998.

Y. Wen and C. Li. A polynomial algorithm for checking diagnosability of Petri nets. *IEEE International Conference on Systems, Man and Cybernetics*, 3:2542–2547, 2005.

M. P. Cabasino, A. Giua, S. Lafortune, and C. Seatzu. Diagnosability analysis of bounded Petri nets. *Proceedings of the 48th IEEE Conference on Decision and Control (CDC) held jointly with 28th Chinese Control Conference*, pages 1254–1260, 2009.

F. Basile, P. Chiacchio, and G. De Tommasi. On k-diagnosability of Petri nets via integer linear programming. *Automatica*, 48(9):2047–2058, 2012.

M. P. Cabasino, A. Giua, and C. Seatzu. Diagnosability of discrete-event systems using labeled Petri nets. *IEEE Transactions on Automation Science and Engineering*, 11(1):144–153, 2014.

B. Liu, M. Ghazel, and A. Toguyéni. Toward an efficient approach for diagnosability analysis of des modeled by labeled petri nets. *Proceeding of the 13th European Control Conference*, 2014a.

S. Haddad, J.-M. Ilié, K. Klai, and F. Wang. Design and Evaluation of a Symbolic and Abstraction-based Model Checker. *2nd International Symposium on Automated Technology for Verification and Analysis (ATVA'04)*, pages 198–210, 2004.

R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

K. Klai and L. Petrucci. Modular construction of the symbolic observation graph. *The* 8th *International Conference on Application of concurrency to System Design*, pages 23–27, 2008.

K. Klai and D. Poitrenaud. MC-SOG: An LTL model checker based on symbolic observation graphs. *Proceedings of the* 29th *International Conference on Application and Theory of Petri Nets*, pages 23–27, 2008.

A. Rushton. A directed graph conainer. *http://www.andyrhshton.co.un/programming/stlplus-library-collection*, 2012.

S. H. Zad, R .H. Kwong, and W. M. Wonham. Fault diagnosis in discrete-event systems: Framework and model reduction. *IEEE Transactions on Automatic Control*, 48(7):1199–1212, 2003.

A. Schumann, Y. Pencole, and S. Thiebaux. Diagnosis of discrete event systems using ordered binary decision diagrams. 15th *International Workshop on Principles of Diagnosis*, 2004.

B. Liu, M. Ghazel, and A. Toguyéni. OF-PENDA: A software tool for fault diagnosis of discrete event systems modeled by labeled petri nets. *International Worshop Petri Nets for Adaptive Discrete-Event Control Systems*, 2014b.

Part II

Session: Program verification

Probabilistic approaches for time critical embedded systems

Liliana Cucu-Grosjean
AOSTE team, INRIA Paris-Rocquencourt
Domaine de Voluceau, BP 105
78153, Le Chesnay
France
liliana.cucu@inria.fr

During the last twenty years different design solutions have been proposed for time critical embedded systems through pessimistic estimation of performances of the processors (thus increased costs) while using average time behavior processors. A possible solution to decrease the pessimism while designing time critical embedded systems is to enrich existing models with appropriate probabilistic descriptions.

time critical embedded systems, probabilistic worst-case reasoning

1. INTRODUCTION

An embedded system is a computing system with a dedicated function, embedded within a larger device,e.g., a defibrillator or an airplane. Today 95% of current processors are embedded, making embedded systems central computing systems of our society. Beside constraints like power consumption and weight, embedded systems may have time constraints and such systems are called time critical embedded systems. Time critical embedded systems design is mainly based on commercial processors with a good average time behavior. During the last twenty years different design solutions have been proposed through pessimistic estimation of performances of the processors (thus increased costs) while using average time behavior processors.

The pessimism of all existing solutions comes mainly from the implementation phase where an absolute value is considered for the worst case execution time of a program. The arrival of modern and more complex processors (e.g., use of caches, multi- and many-core processors) increases the timing variability of programs, i.e., the absolute worst case execution time is becoming significantly larger. For instance, larger execution times require an increased number of processors or more powerful processors.

An intuitive solution to overcome this pessimism is the introduction by Steve Vestal in Vestal (2007) of the notion of mixed criticality for time critical embedded systems. This solution defines several possible values for the worst case execution time of a program on a processor and it has propagated from the original work on scheduling theory Burns and Davis (2015) to synchronous languages Yip and al. (2014), predictable processors Zimmer and al. (2014), model checking Boudjadar and al. (2014), etc. Nevertheless today the mixed criticality solutions are heterogeneous and they are proposed for different phases of design without a common framework.

A possible solution to build such common framework while decreasing the pessimism may be proposed by enriching existing models with appropriate probabilistic descriptions. Probabilistic description of a model provides more information to the designer while allowing several values for a parameter, or several states for a property. Nevertheless, the introduction of probabilities is not trivial as not every probabilistic approach may be used to study time critical embedded systems. First, we prove that the worst case values of the execution times of a program are rare events Cucu-Grosjean and al. (2012). Secondly, the average-case probabilistic reasoning is not useful to guarantee time constraints Maxim and Cucu (2013). We define the probabilistic worst case reasoning as a probabilistic bound on possible values for a parameter or a property of the system Cucu-Grosjean (2013).

In this talk we define probabilistic upper bounds on all possible values or states as the probabilistic

worst case reasoning ensuring the migration of probabilistic methods from modelling soft time constraints to analysing hard time constraints. Two common misconceptions concerning probabilistic time critical embedded systems are discussed: independence and the identical distribution. We summarize recent state-of-the-art research into probabilistic approaches, and we conclude with the main open challenges in this area.

2. DESIGN OF TIME CRITICAL EMBEDDED SYSTEMS

The design of a time critical embedded system may have basically three main phases: (i) the description of the physical process that should be controlled (control theory), (ii) the description of the functional requirements that should be fulfilled (synchronous and asynchronous models) and (iii) the description of the implementation of the time critical embedded system (scheduling or verification).

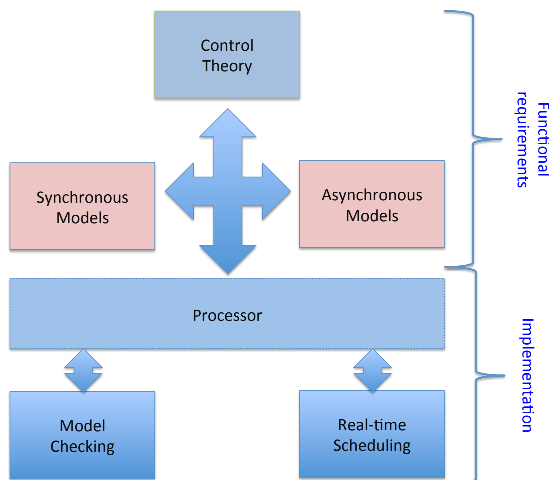


Figure 1: *Different phases of the design of a time critical embedded system*

In order to decrease the pessimism of the design solutions, while ensuring time critical constraints, probabilistic description of parameters may be defined at different levels of design of a time critical embedded system:

- *Probabilistic approaches for control theory for mixed criticality systems*. Solving a control system problem consists in finding the sampling frequency and we identify it as the first property to be described probabilistically.

- *Probabilistic approaches for synchronous models for mixed criticality systems*. The transition between states might be the first property to be described probabilistically by relaxing the synchrony hypothesis.

- *Probabilistic approaches for asynchronous models taking into account mixed criticality systems*. Here the transition between states may be the first to be described probabilistically.

- *Probabilistic approaches for real-time scheduling analysis for mixed criticality systems*.

- *Probabilistic approaches for verification for mixed criticality systems*. The integration of rare events probability distributions in current probabilistic model checking seems to be the first reasonable step.

REFERENCES

Vestal, S. (2007) *Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance* the IEEE Real-Time Systems Symposium.

Burns, A. and Davis, R., (2015) *Mixed Criticality Systems - Review* University of York.

Yip, E. and Kuo, M. and Roop, P. and Broman, D., (2015) *Relaxing the synchronous approach for mixed-criticality systems* the 20th IEEE Real-Time and Embedded Technology and Application Symposium.

Zimmer, M. and Broman, D. and Shaver, C. and Lee, E., (2014) *FlexPRET: A processor platform for mixed-criticality systems* the 20th IEEE Real-Time and Embedded Technology and Application Symposium.

Boudjadar, A.J. and David, A. and Kim, J. and Larsen, K.G. and Mikucionis, M. and Nyman, U. and Skou, A., (2014) *Degree of Schedulability of Mixed-Criticality Real-Time Systems with Probabilistic Sporadic Tasks* Theoretical Aspects of Software Engineering Conference.

Maxim, D. and Cucu-Grosjean, L., (2014) *Response Time Analysis for Fixed-Priority Tasks with Multiple Probabilistic Parameters* the 34th IEEE Real-Time Systems Symposium.

Cucu-Grosjean, L. and Santinelli, L. and Houston, M. and Lo, C. and Vardanega, T. and Kosmidis, L. and Abella, J. and Mezzeti, E. and Quinones, E. and Cazorla, F., (2012) *Measurement-Based Probabilistic Timing Analysis for Multi-path Programs* the 24th Euromicro Conference on Real-time Systems.

Cucu-Grosjean, L., (2013) *Independence - a misunderstood property of and for (probabilistic) real-time systems* Real-Time Systems: the past, the present, and the future.

Towards the Property-Based Testing of an L4 Microkernel API

Cosmin Dragomir
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independentei nr. 313
Sector 6, Bucuresti, Romania
cosmin.dragomir@cti.pub.ro

Lucian Mogosanu
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independentei nr. 313
Sector 6, Bucuresti, Romania
lucian.mogosanu@cs.pub.ro

Mihai Carabas
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independentei nr. 313
Sector 6, Bucuresti, Romania
mihai.carabas@cs.pub.ro

Razvan Deaconescu
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independentei nr. 313
Sector 6, Bucuresti, Romania
razvan.deaconescu@cs.pub.ro

Nicolae Tapus
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independentei nr. 313
Sector 6, Bucuresti, Romania
nicolae.tapus@cs.pub.ro

Software testing has been a significant part of the software development process for the last 30 years and is gaining even more importance with the increasing complexity of software products. As each application has its own requirements, multiple software testing methodologies exist. It is the decision of the developers to choose the best suited types of testing methodologies for their product. This paper presents the design and implementation of a property-based testing framework. Unlike traditional testing methods this methodology uses the formal specification of the API to automatically generate the input and validate the output. The framework will be used to test the API of an L4 microkernel (called VMXL4); VMXL4 possesses the constraints of an embedded environment and of an ongoing development of a stateful system.

Property-Based Testing, L4 Microkernel, API

1. INTRODUCTION

The software industry has been constantly growing in the last decades and the liability and robustness of the software products must match their requirements in order to remain competitive. To obtain a stable product, the entire software stack must be reliable. Therefore software testing must be done at each layer of the software stack, starting with the lowest level: the operating system.

Multiple software testing methodologies are in use nowadays, each of them targeting a degree of test cases coverage and test writing complexity. Alongside the well known unit testing method, another functional methodology named property-based testing has gained ground among software developers. It uses the concept of “tests as specification”, in which tests are written to cover most of the specification.

Writing a large number of tests for the same specification implies a sizable effort from the developers. Property-based testing mitigates this by automaticallly generating the input and creating general and abstract tests known as *properties*. Those can be similar to unit tests, except for the way input is generated and output is validated.

This paper presents an user space framework named QC that is based on an open source basic implementation of a property-based testing framework implemented by Pennebaker (2012). Although the well-known related released frameworks are written in functional programming languages, QC is written in C due to the VMXL4 native environment support.

Porting a new language environment would have meant a sizable and unnecessary effort.

The QC framework solves issues commonly encountered in unit testing by using properties and testing those properties on a large number of randomly generated inputs and by maintaining the internal state of the system. As a downside, QC introduces the problem of formalizing the specification.

The paper is organised as follows: Section 2 is introducing the concepts necessary to understand the paper. Section 3 briefly presents the existing similar frameworks. Section 4 discusses the design and implementation of QC and Section 5 its evaluation in comparison with the existing testing methods for the API of an L4 microkernel, VMXL4. Lastly, in Section 6 we present the overview of the paper and future work.

2. BACKGROUND

This section presents the basic concepts required to understand the next sections of the paper. It starts with the overview and importance of software testing, followed by the essential concepts of unit testing and property-based testing. We show the power of property-based testing in contrast with unit testing. We also do a short introduction of the testing environment, describing the VMXL4 microkernel.

2.1. Software Testing

Nowadays the number of different programming languages, hardware platforms and software libraries is increasing. Requirements, for both specifications and performance, are also more rigorous as time passes by. As a consequence, software applications are becoming more complex and bugs are introduced in new software at all levels. As a countermeasure software testing has gained more importance and attention from programmers.

It is important that applications and services can be stateful or stateless. In a stateful system, an internal state is being held and some of the actions have side effects which might change the state. The design of a stateful software system can be modeled using a finite-state machine and a formal specification of the inputs for every possible state. The summarizing difference between stateful and stateless systems is that for the first one the output depends on the input and possibly on the internal state, whereas for the second one the output always depends only on the input. A well known example of stateful versus stateless are the TCP and UDP transport layer protocols from the TCP/IP stack, where TCP creates and maintains a connection between the client and the server and UDP does not.

At a higher level, software testing can be defined as the process of executing a part or the entire application in order to find errors or to evaluate the quality of the user experience. Any moderately sized application has flaws, but finding them is a complex activity. It is usually unfeasible to do an exhaustive testing on stateful systems, since the number of different possibilities for the input values associated with the existing internal system state is too large. Even for some stateless applications this testing would imply a sizable effort. We conclude that it is more resource and time demanding to test a stateful system instead of a stateless system.

There are multiple methodologies in software testing which must be used in various steps of the development process. In this paper we insist only on those that can be split in two major categories: functional and nonfunctional testing.

Functional testing verifies the client or design specifications by testing the system functionality: checking if the program operations and features behave as they should. In summary it is used to ensure that the application does not have bugs. There are two categories of functional testing: positive and negative. Positive testing is done using valid inputs and comparing the actual output with the expected output, whereas negative testing is done by supplying the system functionalities with invalid or unexpected inputs and operations. In the case of negative functional testing, usually the system must not behave nondeterministically and rather inform the user of the input error.

On the other side, nonfunctional testing is concerned with the user experience, including tests for performance, security, availability, usability. Using this type of testing, one can measure and compare the results in different situations and cover the blanks left by functional testing. For a competitive software product, developers must test the program using both functional and nonfunctional methodologies.

2.2. Unit Testing

One of the most used and successful software testing methodologies is unit testing (Binder (2000); Hunt et al. (2004); Osherove (2010)). It is centered on the concept of unit of work, meaning a single, invokable, logical and functional use case of the system.

Unit testing is composed of a suite of tests that can be run anytime during the development cycle to test certain functions, logic and capabilities of the code. Each test uses a predefined set of inputs, runs a functionality of the system and compares the output with the desired output. If the outputs

differ, the tested functionality has at least one bug. Although unit tests usually verify only one small feature, sometimes it is not easy to find the bug, due to the black-box nature of the methodology. This means that unit testing does not use the internal structure of the functionality, but only the higher level invoked part, and therefore the bug may be in the internal logic and further debugging must be done. One big advantage of unit testing is its reusable nature. Even if the internal logic changes, usually the requirements of the function remains the same and the old test can still be used.

Although unit testing is very useful, it has a very important flaw, which may leave hard detectable bugs in the system. Using unit tests a programmer only verifies a small finite set of inputs and for every different input set he must write another test. Therefore, using unit test programmers cannot even get close to an exhaustive testing. In addition to not finding possible subtle bugs, the work of the programmer is hardened by thinking of all the corner cases and writing more code for them. In the end, these drawbacks are less important than the benefits of unit testing: because it gives good results in practice, it is very used and every major language has frameworks for this testing methodology.

2.3. Property-Based Testing

A software testing methodology which addresses the problems left by unit testing is property-based testing (Fink and Bishop (1997); Fernandez et al. (2004); Machado et al. (2007)). Its main advantage is that it covers substantially more test cases than unit testing; moreover, it can do exhaustive testing though the time required to do this is not directly proportional to the benefits. This is done using only one generic test, named *property*, and some functions to generate the inputs, named *generators*.

A property is the replacement of a unit test and is run multiple times with different automated generated inputs. Every running of the property generates a different test. The input of the property depends on the type and domain specified by the programmer; because the input is generic, the validation conditions of the test must be also generic, following a formal specification. The name "property" comes from the type of test validation, where each test result must pass a general formal property. In layman's terms a property validation condition specifies in a generic way how the tested functionality should behave. A drawback of property-based testing is that the formal specification does not usually exist and the programmer must infer it from the business and logic specifications.

A generator is a callback that does random data generation at each running of the property. Because complicated non-basic data types may be needed, a property-based testing tool must allow user defined generators. In order to achieve a good test coverage, the generated data must be uniformly distributed across its domain.

To show the differences between unit testing and property-based testing, let's assume one would want to test a function named `getMax`, a function which returns the maximum of two integer values. In a unit test he would hard-code two values and test if the output equals to the maximum value. If he wants to test multiple cases, possibly corner cases, then multiple tests need to be written. A pseudocode implementation of the unit test is shown in Listing 1.

```
max = getMax(2, 5)
assert(max == 5)
```

Listing 1: Pseudocode for getMax unit testing

Using property-based testing, a pseudocode implementation would be the one from Listing 2.

```
a = generator()
b = generator()
max = getMax(a, b)
assert((max == a || max == b) &&
       (max >= a && max >= b))
```

Listing 2: Pseudocode for getMax property-based testing

As shown above, the property is generic and more powerful than the unit test. However, the validation condition is bigger and must be correctly determined by the test writer, otherwise the property may give false positives or, even worse, false acceptances.

The quality of the automatic testing tool may be improved by reducing the number of failing test case inputs in order to obtain the minimum set of inputs determining a given failure, a method known as shrinking. This has the advantage of improving the debugging process by providing the programmer with minimal necessary information for debugging; another advantage is that the overhead of this method is not significant.

All in all, property-based testing has the benefits of unit testing and some advantages over it: bigger test coverage, improved specification completeness and it is easier to maintain because of the reduced code size, as illustrated by Nilsson (2014).

2.4. VMXL4

VMXL4 is a general purpose, high performance L4 microkernel (Liedtke (1995)), developed in

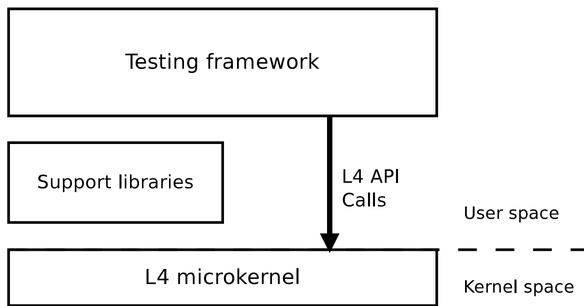


***Figure 1:** VMXL4 testing infrastructure*

partnership with VirtualMetrix, Inc[1]. It provides mechanisms for performance management and a minimal layer of hardware abstraction on which virtualized operating systems personalities can be built. Using the VMXL4 API, trust and security models can be implemented. Examples of systems built using VMXL4 are given in Carabas et al. (2014), Manea et al. (2015) and Mogosanu et al. (2015).

An L4 microkernel was chosen due to the fact that the L4 API's formalization was proven to be feasible by Kolanski and Klein (2006). The seL4 microkernel is the first operating system kernel to be fully formally specified and verified, as shown by Elkaduwe et al. (2008) and Klein et al. (2009). Furthermore, other implementations have been proposed for formally verifiable L4 microkernels (Kauer and Völp (2005)). The property-based testing approach proposed by this paper is in some respects similar to previous work, as it also relies on a formal specification. The most important difference between the two approaches is that property-based testing is more efficient in terms of development resources, as opposed a full mathematical refinement proof, which may require a significant number of man-months to be implemented.

Figure 1 shows the architecture of the current VMXL4 testing infrastructure. The L4 microkernel runs in the privileged CPU mode commonly known as kernel space, while the tests run as user space applications. The testing infrastructure is implemented using support libraries, but the test themselves call the L4 API directly in order to validate it functionally.

Currently most API tests are following unit testing principles, so test coverage is not nearly as extensive. However, microkernels are stateful systems, some of their core mechanisms being strongly coupled. As a result, the current testing framework does not employ true unit tests and only partially validates the interaction between components.

[1] `http://www.virtualmetrix.com/`

We propose that the QC testing framework presented in Section 4 use the same testing design, with the addendum that additional support libraries may be needed, e.g. in order to generate random numbers. This converges with our goal to provide a POSIX compliant native environment based on VMXL4.

3. FRAMEWORKS FOR PROPERTY-BASED TESTING

The idea of a property-based testing framework is not new. Previous frameworks have been developed, but the most successful are for functional languages, due to some of their distinctive features: higher order programming, which is very useful for properties and data generators, lack of side effects, time of development. Moreover, functional programming fits better for random testing than imperative programming because it uses fine-grained properties. This section presents an overview of three of the most influential existing frameworks and of some open source projects.

Haskell QuickCheck

Haskell QuickCheck[2] is the first well known framework for property-based testing and future frameworks were inspired by it. QuickCheck is a tool which automates testing for Haskell programs. As shown in Claessen and Hughes (2002, 2011), it does this by defining a formal specification language, which is powerful enough to represent common forms of specifications: algebraic, model-based and preconditional or postconditional. QuickCheck uses combinators to define properties and test data generators and obtain the test generated data distribution. An important feature of the framework is the shrinking of the generated data when a test fails, to give the minimum input which still fails the property.

Erlang QuickCheck

The programmers who developed Haskell QuickCheck saw the bigger commercial opportunity offered by Erlang and developed a new version of the framework[3], which has its specifications in Erlang. Linking specification in Erlang to code under test in other languages is easier than in Haskell. Two very important distinctive features of the Erlang version are the ability to test stateful systems by using state machine testing and the ability to generate and run parallel test cases in order to find race conditions (Arts and Hughes (2003)).

[2] `https://hackage.haskell.org/package/QuickCheck`
[3] `http://www.quviq.com/products/erlang-quickcheck/`

ScalaCheck
ScalaCheck[4] is the third main framework used for property-based testing and is used for automated randomized property-based testing of programs developed in Scala or Java (Odersky (2010)). It was inspired by Haskell QuickCheck and implements most of its features, but also some in addition to its predecessor, such as stateful testing. Nilsson (2014) provides a comprehensive guide to ScalaCheck.

Open Source Initiatives
Due to the success of Haskell QuickCheck, open source implementations in most major programming languages were started, such as C, C++, Java, Python, but with less features and success. QC was inspired by one of those open source initiatives, employed by Pennebaker (2012).

4. QC DESIGN AND IMPLEMENTATION

This section presents the design and implementation of the QC framework and the motivation behind it. QC intends to test the L4 microkernel API in a functional manner, following the property-based testing methodology. It may be used alongside unit tests, because it tries to generalize them, but on a long term it may strive to replace unit testing for the VMXL4 microkernel API.

4.1. Implementation Starting Point

The implementation is based on the open source project developed by Pennebaker (2012). It is a basic framework, supporting only two features: random data generation and one property per test, which was run for a predefined number of times. A part of the implementation is not really usable, because the programmer who uses the framework must know the size of the generated types and create tests accordingly, which is error-prone. The only part which we partially used is the test data generation component.

The framework is implemented in the C programming language because it was the most convenient option taking into consideration the testing environment. Porting a new language environment can be very complicated, because the native environment offered by a microkernel is very low-level. Moreover, implementing a POSIX environment is equivalent with the implementation of an entire operating system. Also, because the API had already been written in C, there is no need for further linking between different languages.

[4]https://www.scalacheck.org/

4.2. General Design

Because a kernel is a stateful system and QC is developed to test the L4 microkernel API, being evaluated on VMXL4, two more concepts used by QC have to be introduced. Preconditions are a set of predicates that must be true prior to the execution of a property and postconditions are a set of predicates that must be true after the execution of an action in the property. If all the preconditions of a property are true, then the property is applicable, otherwise it is not. If all the postconditions of a property are true, than the property has passed.

Due to the fact that QC is designed for a stateful system, it uses tests containing multiple properties that are used as actions with side effects in the stateful system. Therefore QC borrows some elements from integration testing, a methodology in which individual software modules are tested together. Each test consists of at least one property, randomly generated from the available properties. Each property has a finite number of arguments with known data types at compile time, a fact that provides the opportunity to use property-based testing. When at least one of the postconditions of a property has failed, then the test fails, the entire generic test completes and the actions taken during the test are printed alongside their input. The second situation in which a test fails is when its state becomes inconsistent, meaning that no property has all of its preconditions passing and therefore no future action can be taken. When a test fails, the programmer sees all the randomly generated data used by the test and this facilitates easier debugging.

Properties are divided in two categories: normal properties and cleanup properties. Normal properties are placeholders for actions that the system may take anytime, provided the preconditions are satisfied. Cleanup properties are used to end tests and to free allocated resources. Every test must have at least one cleanup property. If a test does not have allocated resources, QC provides the `empty_clean_property` macro for an empty property, whose only purpose is to end the test. Although most of the time only one cleanup property will be used for a generic test, having multiple cleanup properties may be useful in some situations. One can use fine-grained cleanup properties if the system can have many internal states. This makes the code cleaner and in a system with many generic tests, the probability to reuse cleanup properties is bigger if they are fine-grained.

A higher level design of QC is shown in Figure 2. The programmer must call QC with an array of normal properties and an array of cleanup properties, as previously discussed. To generate random input and

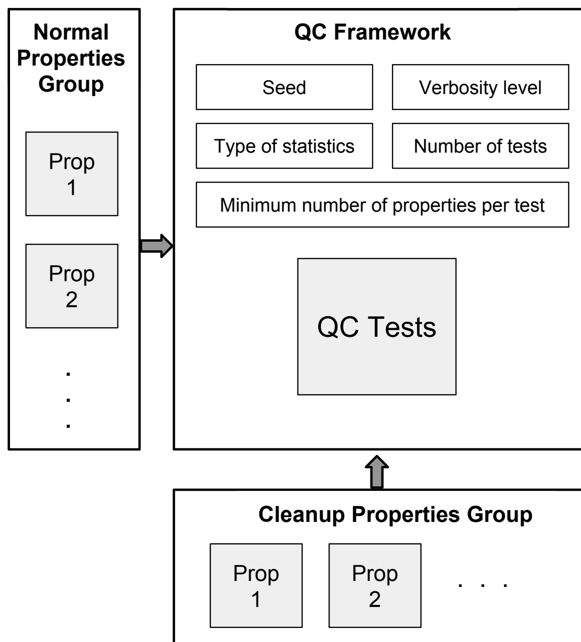


Figure 2: *QC design*

randomly pick properties, QC uses a seed. When a test fails, the programmer will want to reproduce the exact same sequence of properties and inputs to test the fix for the bug. Because the random generation is deterministic given the seed, QC shows it for every generic test so the programmer can use that seed if he wants to reproduce the tests. Otherwise, QC generates a random seed to assure random tests and a good test coverage.

QC will generate a fixed number of tests, previously given by the user. Another available option is the verbosity level for generic tests. The user can see the sequence used by every test or only by failing tests. Viewing the sequence used by every test can be useful to improve the generic test and its test coverage.

There may be cases when tests will end prematurely, after using only a few properties, because QC may choose and use a cleanup property whenever its preconditions are satisfied. To mitigate this, the user chooses the minimum number of properties that must be used during a test.

The last parameter from Figure 2 represents the type of statistics shown at the end of the generic test. QC supports two types of statistics for every generic test: user defined and automatically generated. The user can choose to see both categories, only one or none.

4.3. Generators

As previously mentioned, generators are the callbacks that randomly produce the input data. A QC generator consists of 2 callbacks: one to generate the data and the other to print the data, as the C `printf` function needs different formats depending on the printed type. Because of that, the QC equivalent of a generator is `struct generator_printer`.

QC offers a set of predefined generators for C basic data types, such as `int` and `char`, and also for `bool`, `string` (stored as a `char` dynamic array) and generic arrays. Moreover, a user can write his own generators or printers and use them for his properties.

In order to generate arrays, only the basic type generator is needed, because QC offers a wrapper which automatically generates new array types. The array type can have fixed or random size in a given range, depending on the user needs. All generated arrays are dynamically allocated and their memory is freed after their associated property ends. This avoids out of memory errors for big tests with many generated arrays, but can introduce subtle bugs if the user forgets to copy the content of the array in case he needs it after the property ends.

Sometimes basic types may need additional features such as a maximum or minimum value. QC offers two solutions for this. The first one is that miscellaneous parameters can be added to the generators, in order to modify the generated value to match the requirements. The second solution is to change and update the generated values from the property code. Both solutions are acceptable for code readability, but in general cases the first option should be used, because it's reusable and only the parameters will be changed.

All generated data for a property is stored in a dynamically allocated array with the size in bytes equal to the maximum number of arguments of a property multiplied by the maximum size in bytes of an argument type. This approach solves the problem with the variable number of property arguments and their different types. The value of the generated data is obtained in the property and the user must only know the data type and the index of the argument, something that he had already defined in the state machine test specification.

4.4. Statistics

In order to measure various metrics, QC offers the possibility to attach user defined statistics to a property. After a property ends successfully, each of its statistics callbacks is called and the metrics are updated. This can help the user to investigate bugs and also measure the test coverage. An example of this logging category is shown in Listing 7.

By default, QC logs statistics regarding properties and their sequence. For every property, QC logs the number of total calls, the number of starting test property calls and for every property how many times it followed the current property. An example of this type of logging is shown in Listing 8. The default logging done by QC can be very useful to detect preconditions bugs and see if the tests are surfacing the desired states.

The user decides if he wants to see any of the two statistics category when the QC framework is called to generate and validate tests.

4.5. Properties, Preconditions and Postconditions

Since QC supports stateful system testing, using a property requires the following steps: testing preconditions, getting the values for the randomly generated data, applying actions and testing postconditions.

The preconditions are optional but if they are missing, the property can be always chosen by the framework as the next part of the test. Preconditions are implemented as a callback, differently from the property callback. Because the preconditions depend only on the internal state, not on the generated data, it is better to obtain the new data only if the property can be applied, avoiding generation of useless data, which will be replaced afterwards. Therefore, preconditions must be tested before the data generation step and this can be achieved by having another callback, associated with a property. This approach has another benefit: some preconditions are used by multiple properties and having them as functions gives better reusability. If a property does not have any preconditions, their callback will be `NULL`. To address any possible usage, preconditions can be used from inside the property too, but this is not a good practice, as explained above.

Actions are the main content of properties, because they change the state of the system and their side effects are verified by the postconditions. Actions can be interleaved with their postconditions, which are obtained from the formal specification of the API. As opposed to preconditions, postconditions are located in their corresponding property callback, because they depend on the generated data and we do not obtain a performance improvement if we have them in separate callbacks. Moreover, if the property contains multiple actions, then it is recommended to check the postconditions for an action or a group of actions as soon as possible, in order to have good code readability.

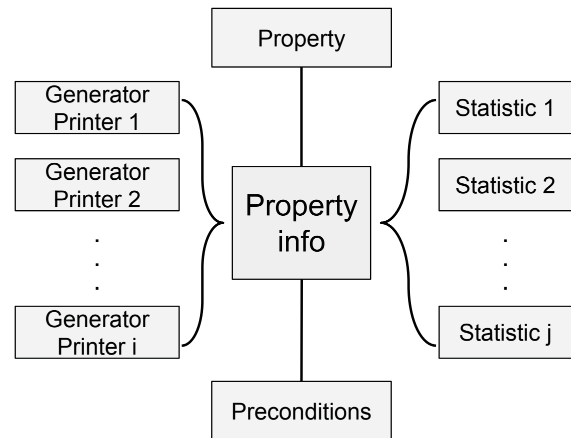


Figure 3: *Property info callbacks*

As can be seen in Figure 3, QC properties are composed of multiple callbacks, stored in a structure named property info. We need an array of generators for the property input and another array of user defined statistics to gather various data. On the other side, we need a callback for preconditions and a callback for the property itself, to make the API calls and test the postconditions. Having all of those callbacks, different components of generic tests become easier to integrate with each other.

```
struct property_info {
    /* callback for the property */
    prop function;

    /* displaying name */
    char const * const name;

    /* array of generators */
    struct generator_printer *gp;

    /* number of generators */
    int gp_size;

    /* preconditions callback */
    precondition prec;

    /* array of statistics callbacks */
    struct user_statistic *stats;

    /* number of statistics callbacks */
    int stats_size;
};
```

Listing 3: Struct property_info

In QC's implementation, property descriptions and callbacks are contained by the `property_info` structure, as shown in Listing 3. In addition to what was previously discussed, the `name` field assigns a

descriptive name to the property and is used for the verbosity option QC_INFO or for failing tests.

4.6. QC test logic

Having all the previously discussed elements, QC can generate and run tests. Figure 4 shows a state machine with the actions taken by QC during a test. Until a test fails or the required number of tests have been run, the framework tries to falsify the generic test by finding a failing test.

Starting a test, QC chooses a property, checks its preconditions (should they exist) and, based on the result, picks another property or continues with the current one. If the preconditions have passed, QC generates test data, executes the actions and the postconditions are checked. If any of the postconditions fails, the testing is over, because QC falsified a sequence of properties. Otherwise, the statistics are updated and if the property is from the cleanup group, the current test completes and another test starts; if the property is from the normal group, the test continues and another property is chosen.

4.7. Pseudorandom number generator

QC has a random module which currently supports two implementations: the POSIX `rand` function and the Mersenne Twister PRNG. The default implementation is Mersenne Twister (presented in Matsumoto and Nishimura (1998)), because it provides better data distribution than `rand` and always has the same output for a given seed, on 32 bits, in contrast with `rand`, whose result may vary depending on the architecture.

4.8. VMXL4 Influence over QC

Internally QC uses a seed for randomizing the test data generation and the chosen property at every step of a test. Because the VMXL4 native environment is under ongoing development and some POSIX functions (e.g. `rand`) are not yet implemented, inside the testing environment the seed is actually a numerical value obtained from a hardware timer provided by the development platform. However, the framework does not depend on a specific platform and is portable, requiring only POSIX functionality.

The VMXL4 API is currently being tested using the Check Unit Testing Framework for C. In order to be as easy as possible to use and because unit tests usually need little changes to become properties, the QC interface has been designed to have some similarities with the Check framework. For that reason, postconditions can be tested using `prop_fail_if` and `prop_fail_unless`, wrappers similar to Check's `fail_if` and `fail_unless`.

5. QC EVALUATION AND TESTING

This section describes evaluation, results and implications, while the framework is still under development. To evaluate the performance of the QC framework, its impact on the test coverage and code size will be detailed.

```
/* normal properties */
struct property_info p[] = {
  {init_property, "init",
    (gp_array){qc_u_int}, 1,
    q_is_not_initialized,
    (stats_array){queue_size_stat}, 1},
  {dequeue_property, "dequeue",
    (gp_array){}, 0,
    q_is_initialized,
    NULL, 0},
  {enqueue_property, "enqueue",
    (gp_array){qc_int}, 1,
    q_is_initialized,
    (stats_array){element_sign_stat}, 1}
};

struct property_group normal_group = {
    .prop = p, .size = 3
};

/* cleanup properties */
struct property_info clean_p[] = {
  {clear_property, "clear",
    (gp_array){}, 0,
    q_is_initialized,
    NULL, 0}
};

struct property_group clean_group = {
  .prop = clean_p, .size = 1
};

qc_for_all(
  /* property groups */
  normal_group, clean_group,
  /* minimum properties */
  5,
  /* verbosity level */
  QC_ERROR,
  /* number of tests */
  1000,
  /* statistics level */
  QC_SHOW_STATS
);
```

Listing 4: QC initialization and calling to test a circular queue library API

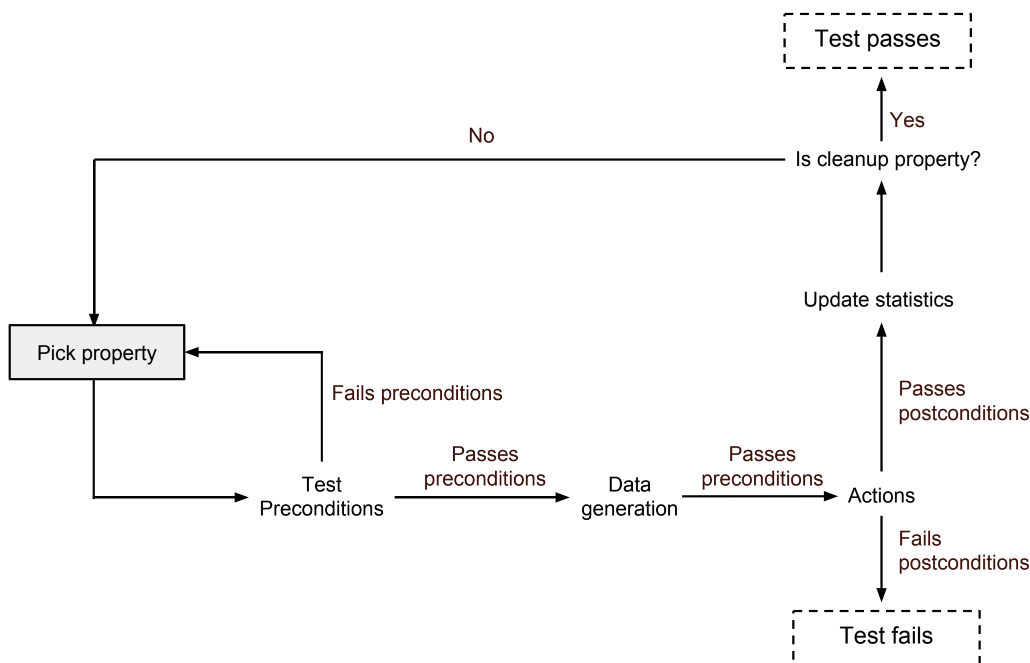


Figure 4: *Test state machine*

QC has been tested so far on two modules. The first module is an implementation of a circular queue, which has been chosen for the following three reasons: it is easier to validate new features of the framework with a simpler module, it is a stateful system, with an internal representation for the queue, and it is a portable module which can be used to validate QC against Haskell QuickCheck. The second module is the thread scheduling module of VMXL4, currently being tested with unit tests using the Check framework.

For the circular queue, the code from Listing 4 was used to initialize QC in order to test the queue library public API. It can be observed that the code uses normal properties and cleanup properties, as mentioned in Section 4. With just four properties, similar to unit tests, the framework automatically generates 1000 test cases with a different number of properties and different sequences of properties, each with automatically generated different inputs. As one can see, there is not much difference in the code logic complexity for unit testing and property-based testing, but the benefits of property-based are significant. When different bugs were introduced on purpose in the queuing logic, QC detected all of them, using only the code from Listing 4 and its four properties.

An example of QC finding a bug for the circular queue is Listing 5. It displays, in order, all the properties taken during the test and their generated input. Judging by the output, it is most likely that there are problems with the enqueue operation when the queue gets full. This is one of the corner cases which the programmer should have taken care of personally if he were to use unit testing.

```
*** Test Failed! ***
Test number 43
________________
init: 2
dequeue:
enqueue: -392470180
enqueue: -692402
```

Listing 5: QC failing test

After solving the bug, QC validates the implementation in Listing 6.

```
+++ Success: passed 1000 tests. +++
```

Listing 6: QC tests passing

For the generic test from Listing 4, the generated user defined statistics are shown in Listing 7. QC displays the total number of statistics for every property. For the `init` property, we wanted to see in what range is the queue size. For the `enqueue` property, we wanted to see the sign of the enqueued number. It can be observed that the numbers are showing a balance for the generated data.

```
——TESTS USER DEFINED STATISTICS——

"init" INFO
total statistics: 1
```

```
    name: "queue_size"
        <20: 201
        <40: 201
        <60: 202
        <80: 203
        <100: 193
--------------------------------------------
"enqueue" INFO
total statistics: 1
    name: "element_sign"
        negative: 1457
        positive: 1423
```

Listing 7: User defined statistics

Last but not least, for the generic test from Listing 4, the QC default statistics are shown in Listing 8. For every property, QC displays the total number of calls, how many times it was the first property of the test and afterwards for every property how many times it followed the current property. Note that for cleanup properties QC doesn't show the following properties, because the cleanup property will be the last from the test. Generally, QC default statistics are useful not only to balance the tests, but also to debug preconditions.

```
---TESTS PROPERTY SEQUENCE STATISTICS---

"init" INFO
total calls: 1000
starting calls: 1000
    init: 0
    dequeue: 527
    enqueue: 473
    clear: 0
--------------------------------------------
"dequeue" INFO
total calls: 3003
starting calls: 0
    init: 0
    dequeue: 1264
    enqueue: 1239
    clear: 500
--------------------------------------------
"enqueue" INFO
total calls: 2880
starting calls: 0
    init: 0
    dequeue: 1212
    enqueue: 1168
    clear: 500
++++++++++++++++++++++++++++++++++++++++++++
"clear" INFO
total calls: 1000
starting calls: 0
```

Listing 8: QC statistics

When porting some of the unit tests for the VMXL4 thread scheduling module to QC properties, the VMXL4 API Reference Manual was needed to understand the behavior of tested functions and to infer the formal specification, which, in the end, is not a sizable effort for a programmer who is accustomed to the design of the module. For some of the ported unit tests the specification was very simple and their content remained almost the same.

Although only a few unit tests were ported to QC properties, the framework already found one inconsistency in the unit test. The faulty unit test was verifying if two threads with different priorities are scheduled accordingly; however on symmetric multiprocessing (SMP) the validation condition was always true. The test would have passed even if the system had a bug.

The inconsistency was found after transforming the unit test into a property and using the same wrong specification. The property was failing, therefore only two causes could have been possible: the property was wrong or the module had a bug. Fortunately, the first case was true and the unit test was the cause. QC found the inconsistency using its random generation feature. This emphasizes that unit tests are not very reliable compared to properties, because usually they do not take into consideration many test cases, therefore they may hide system bugs or even test design bugs.

6. CONCLUSIONS AND FURTHER WORK

Every software system needs testing in order to fulfill its business requirements and, as a consequence, be reliable and successful. This paper concentrates on property-based testing, because although it is more powerful than unit testing, due to its bigger input coverage, it is used less frequently than unit testing. In order to emphasize the property-based testing applicability and importance, the paper gives an overview of the QC framework.

QC is an automated testing tool written in C which runs in the native environment of an L4 microkernel and whose purpose is to test the microkernel API in a functional manner. Because the microkernel is a stateful system, the framework allows the testing of multiple controlled series of operations, besides the usage of random generated input. In order to obtain a thorough testing, QC offers support for generating any data type, using predefined generators which can be combined to obtain new test data generators. To test and evaluate the framework, the native environment of the VMXL4 microkernel is used.

As future work, QC failing tests will be shrinked to a more suggestive failing test, to ease the work of the debugging programmer. Additionally, we aim to analyze QC's code coverage, compare it to that of other L4 testing infrastructures and find ways to improve it.

REFERENCES

T. Arts and J. Hughes. Erlang/quickcheck. In *Ninth International Erlang/OTP User Conference*, 2003.

R. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

M. Carabas, L. Mogosanu, R. Deaconescu, L. Gheorghe, and N. Tapus. Lightweight display virtualization for mobile devices. In *Secure Internet of Things (SIoT), 2014 International Workshop on*, pages 18–25. IEEE, 2014.

K. Claessen and J. Hughes. Testing Monadic Code with QuickCheck. `http://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf`, 2002.

K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the sel4 microkernel. In *Verified Software: Theories, Tools, Experiments*, pages 99–114. Springer, 2008.

J.-C. Fernandez, L. Mounier, and C. Pachon. Property oriented test case generation. In *Formal Approaches to Software Testing*, pages 147–163. Springer, 2004.

G. Fink and M. Bishop. Property-Based Testing; A New Approach to Testing for Assurance. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.5559&rep=rep1&type=pdf`, 1997.

A. Hunt, D. Thomas, and P. Programmers. *Pragmatic unit testing in Java with JUnit*. Pragmatic Bookshelf, 2004.

B. Kauer and M. Völp. L4. sec preliminary microkernel reference manual. 2005.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

R. Kolanski and G. Klein. Formalising the l4 microkernel api. In *Proceedings of the Twelfth Computing: The Australasian Theory Symposium-Volume 51*, pages 53–68. Australian Computer Society, Inc., 2006.

J. Liedtke. *On micro-kernel construction*, volume 29. ACM, 1995.

P. D. Machado, D. A. Silva, and A. C. Mota. Towards property oriented testing. *Electronic Notes in Theoretical Computer Science*, 184:3–19, 2007.

V. Manea, M. Carabas, L. Mogosanu, and L. Gheorghe. Native runtime environment for internet of things. In *Advanced Computational Methods for Knowledge Engineering*, pages 381–390. Springer, 2015.

M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

L. Mogosanu, M. Carabas, R. Deaconescu, L. Gheorghe, and V. G. Voiculescu. VMXHAL: A Versatile Virtualization Framework for Embedded Systems, 2015.

R. Nilsson. ScalaCheck: The Definitive Guide, 2014.

M. Odersky. Contracts for scala. In *Runtime Verification*, pages 51–57. Springer, 2010.

R. Osherove. *The art of unit testing*. mitp, 2010.

A. Pennebaker. A C port of the QuickCheck unit test framework. `https://github.com/mcandre/qc`, 2012.

An Approach for Formal Verification of Updated Java Bytecode Programs

Razika Lounas[1,2]
[1]University of M'hamed Bougara of Boumerdes
Facutly of Sciences, LIMOSE Laboratory
Avenue de l'independance, 35000 Boumerdes
Algeria
[2]University of Limoges
123 Avenue Albert Thomas, 87700 Limoges, France
razika_lounas@umbb.dz

Mohamed Mezghiche
University of M'hamed Bougara of Boumerdes
Facutly of Sciences, LIMOSE Laboratory
Avenue de l'independance, 35000 Boumerdes
Algeria
mohamed-mezghiche@umbb.dz

Jean-Louis Lanet
INRIA LHS-PEC
263 Avenue Général Leclerc, 35000 Rennes
France
jean-louis.lanet@inria.fr

This paper deals with formal specification and verification of Java bytecode update. Programs update for java applications has gained a wide interest since it is used for several purposes: transforming semantics of a program, adding features to a program or performing optimizations. In this paper, we focus on program transformations for java programs at the bytecode level. Because these transformations may introduce errors, our goal is to provide a formal way to verify the update and establish its correctness. Our approach for formal specification and verification of updated Java bytecode programs is based on four ingredients: a formal interpretation of the semantics of update operations, a functional representation of bytecode, bytecode annotation and predicate transformation calculus. We use the concept of Hoare predicate transformation to derive a specification of an annotated bytecode. Annotations are used to express update operations within the code. A functional representation is used to model annotations and bytecode. The approach derives then a new specification for the annotated bytecode using a weakest precondition calculus defined to deal with update operations. Verification conditions are then generated and proved to establish the correction of the update.

Bytecode transformation, formal semantics, weakest precondition calculus, bytecode verification.

1. INTRODUCTION

During their life cycle, programs need to be updated in order to alter their semantics, perform optimizations or add features. Several techniques were presented for this purpose in literature, for example, (Neamtiu et al. (2006) and Gupta et al. (1996)) present systems for C programs updating and (Orso et al. (2002), Hlopko et al. (2013)) present systems to update Java programs.

Updating programs leads to the transformation of their elements such as code, data structures and state. We focus on the transformation of Java codes. In this context, several tools were developed, for example, Java Syntactic Extender (JSE) (Bachrach and Playford (2001)) and ixj (Boshernitsan and Graham (2004)). However, in some cases, the source code is not available (or not distributed). Transforming a program at bytecode level is an interesting alternative since several languages like Java or Java Card are based on virtual machines executing bytecode. Transforming programs at bytecode level offers some advantages: it does not require to recompile which can be a time consuming task as in the case of transformations at source code level. On the other hand, bytecode level transformation is more complex than source-level manipulation for the users because they have to know bytecode language very well and because of the many low-level details one needs to use.

Java bytecode transformation is used in several applications and several tools were developed to manipulate Java bytecode programs such as BCEL

(Dahm (1999)) and RuggedJ (McGachey et al. (2009)). In (Sakamoto et al. (2000)), the authors developed an algorithm to ensure portable thread migration in Java. This algorithm is based on bytecode transformation. Bytecode is transformed in order to enable programs to save and restore their execution state after migration through the network. Another purpose for bytecode transformation is presented in (Binder and Hulaas (2005)) where a framework based on bytecode transformation is developed in order to enable Java applications to perform CPU management.

In some cases, the transformation occurs at runtime. The update is then said to be dynamic (Dynamic Software Update: DSU). In (Noubissi (2011)) and in (Noubissi et al. (2011)), the authors presented a system to perform DSU: while the Java Card virtual machine is executing the program, the bytecode is updated.

This large interest of Java bytecode transformation and its use in many critical applications raise the question of its correctness. In fact, a transformation may introduce an error which may alter the bytecode leading the system to an unexpected state. Besides, in some cases, the update is critical (e.g. EmbedDSU) in such a way that an attacker can take advantage of an incorrect update. In these applications where security issues are involved the update must pass some certification procedure for example Common Criteria (Common Criteria (2015)). For a certain certification level one has to provide a formal proof of the security mechanism implemented. A formal way to specify transformations and verify their correctness is then necessary.

Formal methods offer rigourous means in specifying software properties and establishing the correctness of programs regarding their formal specifications. In this work, we present an approach for formal verification of bytecode update. We focus on Java bytecode and the system presented in (Noubissi et al. (2011)) called embedDSU: a system developed to implement DSU functionalities in Java Card applications. It is based on two parts: off-card in which a module called DIFF generator computes the syntactic changes between the old and the new version of the application and generates a DIFF file (called also a patch). This patch is then sent on the card to perform the update by other modules implemented by extending the Java Card virtual machine.

In this work, we propose to formally verify that the obtained bytecode is semantically equivalent to the one written by the programmer and used to perform the DIFF file. Our approach is based on the following contributions: the definition of a new weakest precondition calculus as the base of the verification process, a formal interpretation of the semantics of the update operations, a functional representation of bytecode programs and bytecode annotation. The choice of functional representation is motivated by our interest in capturing the behavior of the initial bytecode and the updated version and the mature existing tools for formal reasoning about functional programming languages.

This paper is organized as follows: in section 2 we give an overview of embedDSU. Section 3 introduces the language and the formal semantics of the updates. In section 4, we present an overview of our approach in its steps. We present the specification languages is section 5. In section 6, we give our functional modelisation of Java bytecode and annotations. We propose a predicate calculus for update operations in section 7 and give the notion of a correct update. This section ends with an example to show how the logic works. We discuss related work in section 8 and conclude in section 9.

2. OVERVIEW OF EMBEDDSU

EmbedDSU (Noubissi (2011), Noubissi et al. (2011), Noubissi et al. (2010)), is a software-based DSU technique for Java-based smart cards which relies on the Java virtual machine. It is based on the modification of an embedded virtual machine. EmbedDSU is divided in two parts: off-card and on-card:

(i) In off-card, a module called *DIFF generator* determines the syntactic changes between versions of classes in order to apply the update only to the parts of the application that are really affected by the update. The changes are expressed using a Domain Specific Language (DSL). Then, the DIFF file result is transfered to the card and used to perform the update.

(ii) The on-card part is divided into two layers: 1) Application Layer: The binary DIFF file is uploaded into the card. After a signature check with the *wrapper*, the binary DIFF is interpreted and the resulting instructions are transferred to the *patcher* in order to perform the update. The *patcher* initializes data structures for update. These data structures are read by the *updater* module to determine what to update and how to update, by the *safeUpdatePoint detector* module to determine when to apply the update and by the *rollbacker* to determine how to return to the previous version in case of update failure. These points require the introspection of the virtual machine. 2) System Layer: the modified virtual

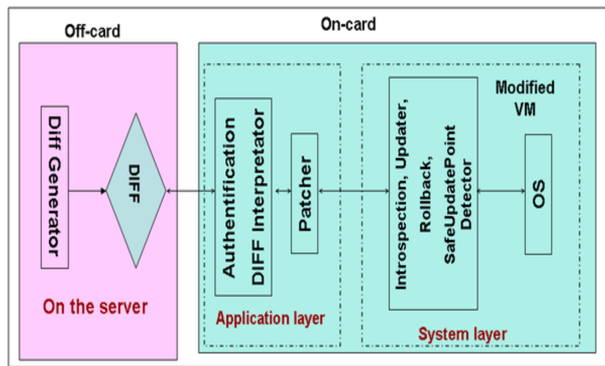


Figure 1: *Architecture of EmbedDSU*

Class compute
int compute_sum(int a, int b) {
int sum =0 ;
sum=a+b ; → **sum =a-b ;**
return sum ;} }
Java file

int compute (int,int) ;
Code
0 :iconst_0
1 :istore_3
2 :iload_1
3 :iload_2
4 :iadd → isub
5 :istore_3
6 :iload_3
7 : ireturn
Bytecode

OxDIFF<class_compute {
Method {
Name : compute_sum
Instr :
Del % 4
Add % isub 4
}end_meth
DIFF file

Figure 2: *An example of a patch (DIFF file)*

machine supports the followings features: (1) *Introspection* module which provides search functions to go through VM data structures like the references tables, the threads table, the class table, the static object table, the heap and stack frames for retrieving information necessary to other modules; (2) *updater* module which modifies object instances, method bodies, class metadata, references, affected registers in the stack thread and affected VM data structures; (3) *SafeUpdatePoint detector* module permits to detect safe point in which we can apply the update by preserving coherence of the system.

The system EmbedDSU is suitable for smard cards especially in term of resource limitations. It was established that sending a DIFF file is less ressource consuming than sendig the whole new version to the card and perform updates and that the resources implied by the update modules are acceptable in term of memory occupation (Noubissi (2011)). The system EmbedDSU updates three principal parts:

(i) The bytecode: the process updates first the bytecode of the updated class and the meta data associated with it e.g., constant pool, fields table, methods table...

(ii) The heap: The process updates the instances of the updated class in the heap, obtains new references for modified objects and updates instances using these references.

(iii) The frames: The process updates in each frame in the thread stack the references of updated objects to point to new instances.

This paper addresses the first part: bytecode update at the method level. The types of updates that may occur are: adding, modifying or suppressing bytecode instructions, changing the signatures of a method or modifying local variables. These updates are contained in the DIFF file which indicates the update and where it occurs in the bytecode. An example is shown figure 2: the patch indicates that the instruction *iadd* in the method *compute_sum* is deleted and the instruction *isub* is added at the same place provided by the program counter.

3. LANGUAGE AND SEMANTICS

3.1. The language

For the definition of the semantics, we extend the formalism used by Freund and Mitchell (Freund and Mitchell (1999)). The authors define a type system for a small subset of Java bytecode. We define a subset and propose to extend it with instructions to indicate updates called update instructions (*Upd_instr*) for instruction addition, deletion and modification. In this definition, x is a local variable; L is an instruction address; A is a class name; f is a field name; l is a method name and pc the program counter.

$$Instruction ::= |pop\ |if\ L\ |store\ x\ |load\ x\ |new\ A$$
$$|binop\ |neg\ |const\ a\ |invokevirtual\ A\ l\ t\ |goto\ L$$
$$|getfield\ A\ f\ t\ |putfield\ A\ f\ t\ |return$$

$$Upd_Instr ::= Add_Inst\ Instruction\ pc$$
$$|Dlt_Inst\ Instruction\ pc$$
$$|Mod_Inst\ Instruction\ instruction\ pc$$

In this language, the instruction pop extracts the top of the stack and $const\ a$ pushes a constant a on the top of the stack. The instruction $load\ x$ pushes the value in the variable x on the top of the stack whereas the instruction $store\ x$ pops the top of the stack and stores it in the variable x. The instruction $if\ L$ jumps to L if the top of the stack is not zero else it performs the following instruction. $Goto\ L$ jumps to L. The instruction $New\ A$ allocates a new object of type A and pushes it on the top of the stack. The instructions manipulating fields are : $getfield\ A\ f\ t$ and $putfield\ A\ f\ t$. Getfield reads the field f, which has the type t of the object of class A whose reference is on the top of the stack and pushes its value on the top of the stack and putfield modifies the field f with the value popped form the stack.

The instruction $invokevirtual$ invokes the method l of signature t and the class A. The instruction $Binop$ is used to gather arithmetic binary operations: add, $mult$ and sub. The instruction neg negates the top of the stack and $return$ is for method return.

Update instructions are respectively: adding an instruction, deleting instruction and modifying an instruction. We indicate the place of the update operation with pc.

3.2. Operational semantics for bytecode instructions

We model the interpretation of the instructions of the bytecode instructions using the standard framework for operational semantics (Freund and Mitchell (1999), Bannwart and Müller (2005)). Each instruction is characterised by the transformation of a configuration. A configuration $< M, s, h, f, pc >$ representing a step execution consists of an operand stack s, a heap h, a local variables map f, a program counter pc and the body M. Operational semantics is defined by a transition relation over configurations. A transition $< M, s, h, f, pc > \rightarrow < M, s2, h2, f2, pc2 >$ takes the state from the configuration $< M, s, h, f, pc >$ to the configuration $< M, s2, h2, f2, pc2 >$.

The rules for the instructions of our language are represented in table 1. The instruction $new\ A$ creates a new object of class A, thereby modifying the current heap. A reference to the new object is pushed onto the stack. $store\ x$ pops a value from the evaluation stack and assigns it to a variable,f is modified accordingly. $load\ x$ put the value of x on the top of the stack. The $binop$ operation which pops two values from the stack, performs the binary operation, and pushes the result. $if\ l$ has two rules; wether it jumps to the indicated line or performs the following instruction according to the value of the top of stack. The instruction $putfield$ updates the heap with the new value of the field of the object which is on the top of the stack. The new value is popped from the second element of the stack. $invokevirtual$ invokes the method l on an object reference and parameters on the stack and replaces these values by the return value v of the invoked method after its execution.

3.3. Formal semantics for update instructions

We propose a static semantics to express the effects of update instructions on a configuration of the bytecode. This semantics was introduced in our initial paper (Lounas et al. (2012)). The purpose of the semantics is to express formally the effects and the conditions of update instructions and thus prevent type errors in the updated bytecode. In this paper, we give more rules and show how to use the semantics to establish that an updated program is well typed. It is also used in further section to derive specifications for program transformations. In the rules shown in tables 2 and 3, F is a mapping from a program point to a mapping from a frame variable to a type. S is a mapping from a program point to an ordered sequence of types, i denotes a program point or an address of code. The map F_i gives a type of local variables at program point i. The string S_i gives the types of entries in the operand stack at program point i. These F and S are useful to our semantics since they contain typing information about valid local variables and entries in the operand stack respectively. SD represents the stack depth and M (mapping) is a function that associates a number to each line. Dom is the set of addresses used by the method. A configuration at line i is represented by $< (F, S, SD, M), i >$. The judgement that expresses that a bytecode BC is well typed by F, S, SD and M is:

$$\frac{\begin{array}{l} F_1 = F_\top, SD1 = 0 \\ S_1 = \varepsilon, M1 = Map(BC) \\ \forall i \in DOM(BC), F, S, SD, M, i \vdash BC \end{array}}{F,S,M,SD \vdash BC}$$

The first two lines of the judgement represent the initial configuration: all variables are mapped to the value top (default initial value), stack depth is zero, the sequence of types is initially empty (ε) and $M1$ is the mapping of the initial bytecode. The last line expresses that each instruction (update instruction) in the bytecode is well typed. This is ensured by the rules given in tables 2 and 3. For illustration, the insertion of the instruction $new\ A$ at line $i + 1$ allows us to obtain a new configuration if the stack depth is incremented, local variables are not affected and in the stack, the type A is inserted. In the instruction $invokevirtual$ the function dom represents the domain of the invoked function (types of its arguments) and the function $card$ represents the number of elements in the domain. The rule expresses that these arguments are popped from the stack of type and then the result is pushed. For the insertion of an instruction representing an arithmetic binary operation $Binop$, we show the rule of the instruction add: this operation pops two elements (integers) from the stack and then pushes the result. $mult$ and sub have analogous explanations by writing the right operation. In the rules, the mapping $M2$ is the result of operations on $M1$. The operations which represent manipulations on bytecode are: *range* and *shift*. The operation *range* extracts from a mapping $M1$ a part $M2$ included between line n and line m. The second operation shifts a part from a mapping between n and m for p positions which is determined by the number of added instructions.

We define the operations $look_for_jumps$ and $update_jumps$ to take into account jumps in bytecode transformation: $look_for_jumps$ returns from a mapping a list of jumps instructions represented by their line number and the operation update_jumps updates jump instructions:
$Look_for_jumps : mapping \rightarrow int\ list$
$Update_jumps : mapping * int\ list * int \rightarrow mapping$

These operations updates jumps within the bytecode if necessary. When we add for instance an instruction at pc, the instructions after this position are shifted and their numbers change. It is then necessary to update goto and if instructions accordingly. These modifications keep the structure of the bytecode coherent. In the rules for instructions suppression (table 3), $Effect_STK$, $Effect_F$ and $Effects_SD$ are used to express the effects of an instruction of the stack and the local variables and stack depth. They are used to readjust these elements to the instruction at $(i+1)$ in the new bytecode after the suppression. The notation $(M2)F$ (Respectively, $(M2)S$) is used to express F (Respectively, S) in the mapping $M2$. We notice that in this formalisation, a modification is considered as a suppression followed by an insertion.

4. APPROACH FOR FORMAL VERIFICATION

The mechanism of EmbedDSU implies the modification of the bytecode of a running application on-card after the conventional verification during the process of its life cycle. In this process, bytecode passes verification process based especially on type verification. The applications of update operations on-card is performed with insertion and suppression of instructions according to the DIFF file. Consequently, we obtain on-card, after the update process, a new bytecode that was not submitted to the conventional verification process. Our framework allows to:

(i) Ensure the validity of update operations of the DIFF file according to the formal specification of the Java Card virtual machine specification.

(ii) Guarantee that the application of the update leads to a bytecode with the specification that is conform to the intended specification (provided by the programmer).

The first point is ensured by the formalisation of the semantics of update operations. In the second point, we aim to establish that given an initial program $P1$, its new version $P2$ and a DIFF file Δ containing the specification of the transformation derived from the differences between $P1$ and $P2$, the application of the DIFF file on-card on $P1$ (noted App_PATCH) leads to $P2'$. The two programs $P2$ and $P2'$ are verified to be

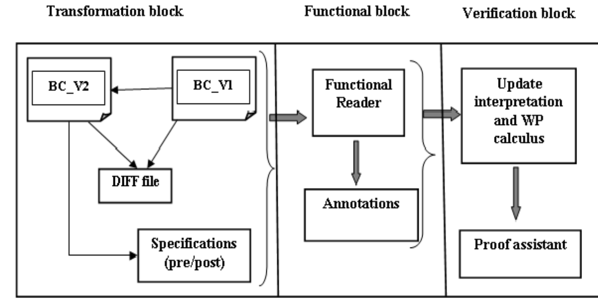


Figure 3: *Approach for verification*

semantically equivalent. This equivalence ensures that the system indeed implemented the desired transformation. This problem can be expressed equationally by:

$\forall P1, P2, P2', \Delta = DIFF(P1, P2), P2' = App_PATCH(P1, \Delta) \Rightarrow P2 \equiv P2'$

This raises two major issues: 1- how to model the application of the DIFF file on an existing program? and 2- how to express the equivalence which guarantees the correctness of the update? We present the overview of our approach for transformation verification. Figure 3 represents an overview of our approach which is split in three parts:

(i) *The transformation block:* in this stage, we obtain from a first version of a bytecode program BC_V1 and a second version BC_V2 (Version one transformed), a DIFF file. This DIFF file will be applied to the on-card first version. We obtain a new version on-card. The goal of our approach is to establish that the on-card new version and BC_V2 are semantically equivalent. At this level, the specifications of both BC_V1 and BC_V2 are provided by the programmer using existing specification languages.

(ii) *The functional block:* we define a functional model for representing and manipulating the Java Card bytecode. We implement an automatic translator called *functional_reader* which takes a program written in bytecode and produces a functional representation of it. The application of the DIFF file is represented at this level as annotations of the functional representation with expressions indicating the place of the update operation and its nature (addition of instructions, deletion . . .)

(iii) *The verification block:* our goal is to verify that the bytecode obtained by transformation is equivalent to the one written by the programmer *i.e.*, it satisfies the same specification. The

Table 1: *Rules for operational semantics*

$$\frac{M[pc]=pop}{<M,v.s,h,f,pc>\rightarrow\ <M,s,h,f,pc+1>}$$

$$\frac{M[pc]=new\ A,h'=h[create(A,ref)]}{<M,s,h,f,pc>\rightarrow\ <M,ref.s,h',f,pc+1>}$$

$$\frac{M[pc]=load\ x}{<M,s,h,f,pc>\rightarrow\ <M,f[x].s,h,f,pc+1>}$$

$$\frac{M[pc]=store\ x}{<M,v.s,h,f,pc>\rightarrow\ <M,s,h,f[x\leftarrow v],pc+1>}$$

$$\frac{M[pc]=if\ l}{<M,0.s,h,f,pc>\rightarrow\ <M,s,h,f,pc+1>}$$

$$\frac{M[pc]=if\ l,v\neq 0}{<M,v.s,h,f,pc>\rightarrow\ <M,s,h,f,l>}$$

$$\frac{M[pc]=const\ a}{<M,s,h,f,pc>\rightarrow\ <M,a.s,h,f,pc+1>}$$

$$\frac{M[pc]=getfield\ \ a\ f\ t,v=h[o.f]}{<M,o.s,h,f1,pc>\rightarrow\ <M,v.s,h,f1,pc+1>}$$

$$\frac{M[pc]=neg}{<M,v.s,h,f,pc>\rightarrow\ <M,(-v).s,h,f,pc+1>}$$

$$\frac{M[pc]=binop,op\in\{+,-,*\}}{<M,v1.v2.s,h,f,pc>\rightarrow\ <M,(v1\ op\ v2).s,h,f,pc+1>}$$

$$\frac{M[pc]=putfield\ A\ f\ t,h'=h[o.f\leftarrow v]}{<M,o.v.s,h,f1,pc>\rightarrow\ <M,s,h',f1,pc+1>}$$

$$\frac{M[pc]=goto\ l}{<M,s,h,f,pc>\rightarrow\ <M,s,h,f,l>}$$

$$\frac{M[pc]=invokevirtual\ A\ l\ t\ ,<l,\varepsilon,h,f_l,0>\rightarrow<l,v,h1,f_l',pc_l>}{<M,a_1...a_n.s,h,f,pc>\rightarrow\ <M,v.s,h1,f1,pc+1>}$$

Table 2: *Rules for update operations (insertion of instructions)*

$$\frac{\begin{array}{l} Add_inst\ \ goto\ L(i+1) \\ SD_{i+1}=SD_i\ PC_MAX++ \\ S_{i+1}=S_i\ \ F_{i+1}=F_i \\ M2= \\ Add_inst(M1,goto\ L,i+1) \\ i+1,L\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ \ store\ x(i+1) \\ SD_{i+1}=SD_i-1\ \ PC_MAX++ \\ S_i=t.S_0\ \ F_{i+1}=F_i[x\leftarrow t] \\ S_{i+1}=S_0 \\ M2=Add_inst(M1,store\ x,i+1) \\ i+1\in DOM(BC)\ x\in VAR(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ add(i+1) \\ SD_{i+1}=SD_i-1 \\ S_i=int.int.S_0\Rightarrow \\ S_{i+1}=int.S_0 \\ M2=Add_inst(M1,add,i+1) \\ i+1\in DOM(BC)\ \ F_{i+1}=F_i \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ \ pop\ (i+1) \\ SD_{i+1}=SD_i-1\ \ F_{i+1}=F_i \\ S_i=t.S_0\rightarrow S_{i+1}=S_0 \\ M2=Add_inst(M1,pop,i+1) \\ PC_MAX++ \\ i+1\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ putfield(A,f,t)(i+1) \\ SD_{i+1}=SD_i-2\ F_{i+1}=F_i \\ S_i=t.A.S_0\Rightarrow S_{i+1}=S_0 \\ M2= \\ Add_inst(M1,putfield(A,f,t),i+1) \\ PC_MAX+3\ \ \ i+1\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ \ new\ A(i+1) \\ SD_{i+1}=SD_i+1 \\ S_{i+1}=A.S_i\ \ \ F_{i+1}=F_i \\ M2= \\ Add_inst(M1,new\ A,i+1) \\ PC_MAX++ \\ i+1\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ getfield(A,f,t)(i+1) \\ SD_{i+1}=SD_i \\ S_i=A.S_0\Rightarrow S_{i+1}=t.S_0 \\ M2= \\ Add_inst(M1,getfield(A,f,t),i+1) \\ PC_MAX+3\ \ \ \ F_{i+1}=F_i \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ \ load\ x(i+1) \\ SD_{i+1}=SD_i+1 \\ PC_MAX++ \\ S_{i+1}=F_i[x].S_i\ \ F_{i+1}=F_i \\ M2= \\ Add_inst(M1,load\ x,i+1) \\ i+1\in DOM(BC)\ x\in VAR(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ \ if\ L(i+1) \\ SD_{i+1}=SD_i \\ PC_MAX++ \\ S_{i+1}=S_i\ \ F_{i+1}=F_i \\ M2= \\ Add_inst(M1,if\ L,i+1) \\ i+1,L\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ \ invokevirtuel(A,l,t)(i+1) \\ SD_{i+1}=SD_i-(card(dom(t))+1) \\ S_{i+1}=tn_1.tn_2\dots tn_n.S_0\rightarrow S_{i+1}=S_0 \\ M2= \\ Add_inst(M1,invokevirtuel(A,l,t),i+1) \\ i+1\in DOM(BC)\ \ F_{i+1}=F_i \\ PC_MAX+3 \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ \ const\ a(i+1) \\ SD_{i+1}=SD_i+1 \\ PC_MAX++ \\ S_{i+1}=int.S_i\ \ F_{i+1}=F_i \\ M2= \\ Add_inst(M1,const\ a,i+1) \\ i+1\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ \ neg\ (i+1) \\ SD_{i+1}=SD_i\ F_{i+1}=F_i \\ S_i=int.S_0=S_{i+1} \\ M2= \\ Add_inst(M1,negi+1) \\ PC_MAX++ \\ i+1\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

Table 3: *Rules for update operations (suppression of instructions)*

$$\frac{\begin{array}{l} Dlt_inst \;\; goto\ L \;\; (i+1) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, goto\ L, i+1) \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1, L \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst \;\; (store\ x \;\; (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, store\ x, i+1) \\ S_i = t.S_0, F_i[x] = t \rightarrow \\ (M2)S_{i+1} Effects_STK(M2[i+1], t.S_0) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst \;\; (add \;\; (i+1)) \\ M2 = Dlt_inst(M1, add, i+1) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ S_i = int.int.S_0 \rightarrow \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst \;\; (pop \;\; (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, pop, i+1) \\ S_i = t.S_0 \rightarrow \\ (M2)S_{i+1} = Effects_STK(M2[i+1], t.S_0) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst \;\; (putfield(A, f, t) \;\; (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, putifield(A, f, t), i+1) \\ S_i = A.t.S_0 \rightarrow \\ (M2)S_{i+1} Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst \;\; (getfield(A, f, t) \;\; (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, getifield(A, f, t), i+1) \\ S_i = A.S_0 \rightarrow \\ (M2)S_{i+1} Effects_STK(M2[i+1], A.S_0) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst \;\; new\ A \;\; (i+1) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, new\ A, i+1) \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst \;\; if\ L \;\; (i+1) \\ SD_i = a \rightarrow SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, if\ L, i+1) \\ S_i = int.S_0 \rightarrow \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1, L \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst \;\; (neg \;\; (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, neg, i+1) \\ S_i = int.S_0 \rightarrow \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst \;\; (load\ x \;\; (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, load\ x, i+1) \\ (M1)S_{i+1} = t.S_0 \rightarrow \\ (M2)S_{i+1} Effects_STK(M2[i+1], S_0) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst \;\; (const\ a \;\; (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, const\ a, i+1) \\ S_i = S_0 \rightarrow \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst \;\; (invokevirtuel(A, l, t) \;\; (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, invokevirtuel(A, l, t), i+1) \\ S_i = tn_1.tn_2 \ldots tn_n.S_0 \rightarrow \\ (M2)S_{i+1} Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F, S, M2, SD, i+1 \vdash BC}$$

specification of the obtained bytecode in its functional representation with annotations is performed by a weakest precondition calculus that we define specially to deal with update operations. A verification condition generator gives then statements to be verified to establish that the obtained specification matches the specification given by the programmer at the level one. A proof assistant is used to discharge verification conditions.

```
int compute (int,int) ;
Code
0 :iconst_0
1 :istore_3
2 :iload_1
3 :iload_2
4 :iadd
5 :istore_3
6 :iload_3
7 : ireturn
```

```
OxDIFF<class_compute {
Method {
Name : compute_sum
Instr :
Del % 4
Add % isub 4
}end_meth
```

```
int compute(int,int) ;
Code
0 :iconst_0
1 :istore_3
2 :iload_1
3 :iload_2
//* Del 4
//* Add isub 4
4 :iadd
5 :istore_3
6 :iload_3
7 : ireturn
Annotated Bytecode
```

Figure 4: *Bytecode annotation with update instructions*

5. JML AND BML SPECIFICATIONS

The starting point is a new version BC_V2 of un existing program BC_V1. First the programmer writes the new version with its specification in terms of pre/post conditions. The specification language used is JML (Java Modeling Language).

JML (Burdy et al. (2005)) is a specification language for Java/Java Card programs. It allows assertions to be included in the source code, specifying for example pre- and postconditions and invariants. JML annotations are a special kind of Java comments: they are preceded by / / @, or written between /* @ and @* /.

A simple method specifications is of the form:

```
/*@ normal_behavior
requires : <precondition> ;
ensures : <postcondition> ;
@*/
```

This specification means that if the precondition (*requires*) holds at the beginning of a method invocation, then the method terminates normally and the postcondition (*ensures*) will hold at the end of the method. Constructs are defined to write assertion such as: $\backslash old$, to denotes the old value of a variable,$\backslash result$ to denote the result of a method and the quantifiers, $\backslash forall$ and $\backslash exists$.

The DIFF file in the system EmbedDSU is created from the program's bytecode. To ensure the correctness of the transformation, the verification of the specification will be done at bytecode level. The language BML (Burdy et al. (2007)), allows to express specifications of bytecode programs. Its formalism is based on JML and the structures of specifications in both languages are very similar.

At the transformation block, specifications for both first version and second version are written in JML. Starting from a specified source code $\{P_{jml}\}code_{source}\{Q_{jml}\}$, with P_{jml} and Q_{jml} representing respectively precondition and postcondition of $code_{source}$, we obtain a specified bytecode program $\{P_{bml}\}code_{BC}\{Q_{bml}\}$. This information is obtained by applying a compiler JML2BML and will be used by the next stages of the approach to perform verification condition generation and ensure the transformation correctness.

6. ANNOTATION AND FUNCTIONAL REPRESENTATION OF BYTECODE

The DIFF file containing the update instructions is calculated at bytecode level and then sent to perform the update on-card. In order to ensure that we send the right one, we model its application on an initial version of bytecode P1 as annotations. The operation of annotating a bytecode with expressions indicating where an update instruction occurs and what is the operation involved can be defined recursively as an annotation function which transforms a program to an annotated program.

$Annot(\varepsilon, P) \equiv P$

$Annot([Upd_i|\Delta], P) \equiv let\ P' = Add_Annot_Line(Upd_i, P)\ in\ Annot(\Delta, P')$

The annotation of a program with an empty DIFF file (ε) is the program itself otherwise, the function iterates over the update operations (Upd_i) in the patch and adds a corresponding annotated line ($Add_Annot_Line(Upd_i, P)$) to the program. Figure 4 shows an annotated program obtained by the application of a DIFF file on an initial byte code. The annotations are represented as special commentaries. For example, *Del 4* : deletes the instruction at program counter (pc) 4 and *add isub 4*, adds the instruction *isub* at pc 4.

In our framework, we use a functional representation for both bytecode programs and annotation function. Figure 5 shows a fragment of the formalisation written in OCaml. We start by defining the data manipulated by the program (integers, objects and variables, then, we formalise the instructions of the sub language. The definition of an instruction is given by the name of a construct (representing the name of the instruction) followed by its arguments. For example, for the instruction *new*, we have the construct *New* taking an *Object* as argument and the instruction *putfield* is represented by the construct *Putfield* followed by a triple representing

```
♯ type object = Object;;
♯ type variable = Var;;
♯ type integer = Int;;
♯ type types = V of variable | O of object | I of integer;;
type jc_instr =
pop
| Store of variable
| Load of variable
| Const of integer
| Add
| Neg
| Return
| New of object
| If of integer
| Goto of integer
| Invoke of object * string * string
| Putfield of object * string * string
| Getfield of object * string * string;;
type bc_line = Line of int *jc_instr;;
type bc = Bc of bc_line list;;
type annot_line= AnnotL of bc_line * string;;
type annot_bc = AnnotBC of annot_line list;;
...
```

Figure 5: *An extract of functional modelisation of bytecode*

the arguments: the class (*Object*) and the names of the type of the field and its name as strings.

A bytecode line is defined as a number (representing the program counter) with an instruction. The bytecode is represented as a list of bytecode lines. An annotated line is represented by the product of a bytecode line and a string representing the annotation. An annotated bytecode is a list of annotated bytecode lines. The result of this modelisation is used to derive specifications of updated programs.

7. VERIFICATION

Our approach for verification is based on the fact that the transformation of a bytecode (of its semantics) implies the transformation of its specification. In Hoare Logic (Hoare (1969)), a program $P1$ and its specification is represented by a triple $\{pre1\}P1\{post1\}$ where $pre1$ ($post1$) is the precondition (postcondition) of the program $P1$. A new version of this triple written off-card by the programmer is $\{pre2\}P2\{post2\}$ (a target triple). The DIFF file is performed with P1 and P2 and then sent to the card to perform update operations, meaning, obtaining a new bytecode and a new spacification. Our goal is to establish that the target triple and the obtained triple match.

7.1. Interpretation of the update

In order to formally define our update interpreter, we need to define some notions. In this interpretation, a state is modeled by a 3-tuple:$< Heap, Frame, Stack - Frame >$ which represents the machine state where Heap represents the contents of the heap, Frame represents the execution state of the current Method and, Stack-Frame is a list of frames corresponding to the call stack. A frame contains the following elements : the stack of operands $OperandStack$ and the values of the local variables $LocalVar$ at the program point PC of the method $Method$ ($< H, Method, PC, OperandStack, LocalVar >$). The definition of the update interpretation is based on the notion of step.

Definition 1. ***Step*** The semantics of an instruction (update instruction) is specified as a function step: $Bytecode_Prog * State * Specification- > State * StepName * Specification$ that, given a bytecode P, a state S and a specification SP, computes the next state S', the name of the next step and a new specification.

Definition 2. ***Java bytecode update interpreter*** We define now an update interpreter (Upd_int) which iterates over steps, take as parameters an annotated program in its functional representation, an initial state and an initial specification and relies on predicate calculus and update interpretation function to produce a new state and a new specification. The interpreter is defined as $Upd_int(BC, S) = (S', Sp')$ with $S = initial(BC, Sp)$ the function for defining an initial state for the execution of the bytecode BC with the initial specification Sp. The Code BC is given with its parameters and an initial heap. The result of the interpreter is a state S' and a new specification Sp'.

Definition 3. ***Verified updated bytecode***

- Let $P1\ and\ P2$ be the first and the new version of a program and P a patch,
- let $P2' = annot(P1, P)$ be the program obtained by annotation of P1 with P,
- let $f(P2')$ the functional representations of $P2'$,
- let $spec(P1) = (pre1, post1)$ the specification of $P1$ and $spec(P2) = (pre2, post2)$ the specification of P2,

We say that $P2'$ is a successfully verified update of $P1$ if and only if: $verification(spec(P2), spec(P2'))$ succeeds where $spec(P2')$ is obtained by predicate transformation on $f(P2')$ starting from $post2$.

7.2. Weakest precondition calculus

In this section, we define a bytecode update logic in terms of a weakest precondition calculus. The proposed weakest precondition (WP) considers that each (update) instruction has a precondition. An instruction with its precondition is called an

Table 4: *Defining rules for weakest precondition calculus for update operations*

wp(Add_instr(pop,i)) $= (shift_exp^2(@E_i))$
wp(Add_instr(store x,i)) $= shift_exp^2(@E_i)(S(0)/x)$
wp(Add_instr(if L,i)) $= ((S(0) = 0) \Rightarrow shift_exp^2(E_L)) \wedge ((S(0) \neq 0) \Rightarrow shift_exp^2(@E_i))$
wp(Add_instr(load x,i)) $= unshift_exp(shift_exp(@E_i))(x/S(0))$
wp(Add_instr(const a,i)) $= unshift_exp(shift_exp(@E_i))(a/S(0))$
wp(Add_instr (new A,i)) $= unshift_exp(shift_exp(@E_i[create(H, A)/S(0), A :: H/H])$
wp(Add_instr(add,i) $= (shift_exp^2(@E_i))[(s(1) + S(0))/S(1)]$
wp(Add_instr(neg,i) $= (unshift_exp(@E_i))[-S(0)/S(0)]$
wp(Add_instr (getfield a f t,i)) $= shift_exp(@E_i[(val(S(0), (a, f)))/S(0)]) \wedge S(0) \neq null$
wp(Add_instr(putfield a f t,i)) $= (shift_exp^3(@E_i))[H((S(0), (a, f)) := S(1))/H] \wedge S(0) \neq null$
wp(goto l1) $= shift_exp(E_{l1})$

instruction specification and is noted as: $E_i : I_i$ where I_i is the instruction and the expression E_i its specification. This notation expresses that the precondition E_i holds when the program pointer is at the program counter *i*. Table 4 shows the calculus of the WP rules for the update operations (inserting instructions).

Functions and notations used. The functions $shift_exp$ and $unshift_exp$ are used to express: the effect of pushing (popping) elements to (from) the stack S and the effect of shifting an expression regarding to the stack elements due to the insertion of instructions. They are defined as follows:

$shift_exp(Exp) = Exp[s(i + 1)/s(i)\ forall\ i \in N]$
$unshift_exp = shift_exp^{-1}$

The elements of the stack are represented by positive integers, the top stack is 0. The symbol $@$ is used to express the old specification associated to a position i: when we add an instruction at position i, the program and the specification are shifted from i and then a new instruction is inserted. Its precondition is calculated with the specification of the instruction that was at position i before the update.

In the rules, for the instructions $store\ x$, $load\ x$, and pop, a precondition is obtained, as in Hoare's assignment (Hoare (1969)) by substituting the right-hand side by the left-hand side in the postcondition. The precondition of an instruction $store\ x$ under a postcondition E_{i+1} (the precondition of the following instruction) is given by: $shift_exp(E_{i+1})(S(0)/x)$ meaning that if the expression E holds after the execution of $store\ x$ then it also holds for the top of the stack before storing it in x. The function $shift_exp$ is used to express that before the execution of the instruction, the top of the stack corresponding to the instruction at $i + 1$ was at index 1.

Inserting an instruction, *e.g.* $store\ x$ at line i means that the precondition of the old instruction at i becomes the postcondition of the inserted instruction and thus the calculated precondition starts from this old postcondition ($@E_i$). The function $shift_exp$ is used twice ($shift_exp^2$) to express also the impact due to the insertion of the instruction on the specifications of the following instructions.

The instructions $new, putfield$ and $getfield$ are heap manipulating instructions. The function $create$ used in the instruction $new\ A$ returns a new object of type A in the heap H. This obtained heap $(A :: H)$ replaces the old heap. The function val used in the definition of getfield to get the value of the field f of the class a from the address (top of the stack). This value is then pushed on the stack. In $putfield$, the value of the field designated by the top of the stack is updated with the value at the second elements of the stack. The insertion of this instruction which pops two values implies three applications of $shift_exp$.

In order to establish semantical equivalence of a code written by the programmer and a program obtained by applying a DIFF file, we check the equivalence of the weakest precondition of an annotated program obtained by WP calculus and a precondition written by the programmer before DIFF file is performed.

7.3. Example

In order to illustrate how the logic works, we take the example of the function abs that returns the absolute value of an integer taken as argument. This function is then transformed in order to get the double of the result in the initial calculus: for an integer given as argument, the new function returns the abstract value multiplied by two (modified abs). The specifications of the two functions are respectively:

$\{p = P\}\ abs\ \{(P \geq 0 \rightarrow result = P) \wedge (P < 0 \rightarrow result = -P)\}$

$\{p = P\}\ modified\ abs\ \{(P \geq 0 \rightarrow result = 2 * P) \wedge (P < 0 \rightarrow result = -2 * P)\}$

In the specification, P denotes the logical value at the entry and $result$ is the result of the function. Figure 6 shows the bytecode of the first version (a) and the second version (b) of the described function. The part (c) of the figure shows the DIFF file generated from the two versions. The last part of the figure (d) shows the bytecode of the function abs annotated with update instructions. We notice that in this bytecode local variables are represented by integers: in $load\ 1$ for example, the number 1 means the local variable 1. The same notation is applied to other local variables.

In figure 7, The WP calculus is performed on the bytecode (without annotation) starting from the postcondition of the new version. The WP calculus is applied on the annotated bytecode as shown on figure 8. The specification for the update instructions are in bold. This example shows that we obtain the same precondition $\{P = v0\}$ which means that at the beginning of the calculus the logical value P is in the first local variable of the function. This result expresses the equivalence of the two bytecodes according to our definition of verified updated program.

8. RELATED WORK

Several studies have been conducted in order to use formal semantics to prevent type errors in bytecode. Our work extends the formalism presented in (Freund and Mitchell (1999)). This work defined semantics and a type system to study object initialization in bytecode. The original idea was developed in (Stata and Abadi (1999)) to study bytecode subroutines. In (Freund and Mitchell (2003)), the authors extended the work (Freund and Mitchell (1999)) to bytecode subroutines, virtual method invocation and exceptions. On another side, using predicate transformation to reason about bytecode properties has been studied in (Grégoire,Sacchini and Sivan (2008)). The authors presented a verification condition generator for bytecode formalized in the Coq proof assistant and based on weakest precondition calculus. Another work using weakest precondition to generate verification conditions from an annotated bytecode is presented in (Burdy and Pavlova (2006), Burdy et al. (2007)).

Our work is close to (Freund and Mitchell (1999)) in the sense of the use of static semantics to analyze bytecode. The specificity of our work is the definition of semantics for updates. We use predicate transformation to reason about bytecode properties using existing tools for specification and proofs. Our bytecode logic for weakest precondtion calculus is inspired by (Bannwart and Müller (2005)). The authors present a Hoare-style logic combined with instruction specification in term of precondition for sequential bytecode. We adopted such instruction specification in our logic for weakest precondition for update operation.

In some studies, manipulating and analysing bytecode requires its modelisation in flexible representations suitable to the manipulation required. In (Puder and Lee (2009)), bytecode is represented by XML trees in order to use the technologies supporting XML to ease the injection and extraction of bytecode. In (Albert and al. (2007)), bytecode is represented by clauses written in Prolog to perform verification of bytecode programs. Generally, functional modelisation is used when the goal is to consider programs as mathematical models whose meaning is independent of runtime states. Therefore, it is possible to apply equational rewriting and reasoning to them (Guodong (2010)) and use several proof systems that are built on or uses functional languages in specifications.

9. CONCLUSION

In this paper, we proposed an approach for formalisation and verification of java bytecode updated programs. Our approach relies on four main concepts. We showed first how to use existing specification languages for Java and Java bytecode programs to write specification and transform them. Then, we defined a formal semantics which constitute a formal mean to establish the validity of update operations with regard to Java type safety. We proposed a functional representation of bytecode in order to model the application of update operations with the use of the notion of bytecode annotation. We presented a predicate transformation calculus based on weakest precondition for update operations to derive a specification for the annotated bytecode and showed how to establish the correctness of the update.

The approach presented is implemented using the OCaml language. Our study started with considering the system EmbedDSU but this is not restrictive, the framework proposed can be generalised to specification and verification of updated programs written in languages that are complied to bytecode. The use of the functional language and representation eases its integration with existing formal methods. Our immediate future work is to define WP calculus for instruction suppression. We plan to define another predicate transformation calculus (strongest postcondition) for update operation and the integration of our approach in an existing formal method supporting verification condition generation for functional programs.

```
int abs(int);
0: load 0
1: if 5
2: load 0
3: store 1
4: goto 8
5: load 0
6: neg
7: store 1
8: load 1
9: return
```
(a)

```
int abs(int);
0: load 0
1: if 7
2: const 2
3: load 0
4: mul
5: store 1
6: goto 12
7: const 2
8: load 0
9: neg
10: mul
11: store 1
12: load 1
13: return
```
(b)

```
OxDIFF<class_compute>
Method {
Name : abs
Instr :
Add const 2 2
Add mult 4
Add const 2 5
Add mult 7
} end_meth
```
(c)

```
int abs(int);
0: load 0
1: if 5
// Add const 2 2
2: load 0
// Add mul 4
3: store 1
4: goto 8
// Add const 2 5
5: load 0
6: neg
// Add mul 7
7: store 1
8: load 1
9: return
```
(d)

Figure 6: *An example of an annotated bytecode (abs)*

int abs(int);
0: load 0 $\{(P = v0\}$
1: if 7 $\{(P \geq 0 \rightarrow 2 * v0 = 2 * P) \wedge (P < 0 \rightarrow 2 * v0 = -2 * P)\ \}$
2: const 2 $\{(P \geq 0 \rightarrow 2 * v0 = 2 * P) \wedge (P < 0 \rightarrow 2 * v0 = -2 * P)\ \}$
3: load 0 $\{(P \geq 0 \rightarrow s(0) * v0 = 2 * P) \wedge (P < 0 \rightarrow s(0) * v0 = -2 * P)\ \}$
4: mul $\{(P \geq 0 \rightarrow s(1) * s(0) = 2 * P) \wedge (P < 0 \rightarrow s(1) * s(0) = -2 * P)\ \}$
5: store 1 $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
6: goto 12 $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
7: const 2 $\{(P \geq 0 \rightarrow 2 * (-v0) = 2 * P) \wedge (P < 0 \rightarrow 2 * (-v0) = -2 * P)\ \}$
8: load 0 $\{(P \geq 0 \rightarrow s(0) * (-v0) = 2 * P) \wedge (P < 0 \rightarrow s(0) * (-v0) = -2 * P)\ \}$
9: neg $\{(P \geq 0 \rightarrow s(1) * (-(s0)) = 2 * P) \wedge (P < 0 \rightarrow s(1) * (-(s0)) = -2 * P)\ \}$
10: mul $\{(P \geq 0 \rightarrow s(1) * s(0) = 2 * P) \wedge (P < 0 \rightarrow s(1) * s(0) = -2 * P)\ \}$
11: store 1 $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
12: load 1 $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
13: return $\{(P \geq 0 \rightarrow result = 2 * P) \wedge (P < 0 \rightarrow result = -2 * P)\ \}$

Figure 7: *WP calculus on the modified function*

int abs(int);
0: load 0 $\{(P = v0\}$
1: if 5 $\{(P \geq 0 \rightarrow 2 * v0 = 2 * P) \wedge (P < 0 \rightarrow 2 * v0 = -2 * P)\ \}$
// Add const 2 2 **{(P ≥ 0 → 2*v0 = 2*P) ∧ (P < 0 → 2*v0 = -2*P) }**
2: load 0 $\{(P \geq 0 \rightarrow s(1) * v0 = 2 * P) \wedge (P < 0 \rightarrow s(1) * v0 = -2 * P)\ \}$
// Add mul 4 **{(P ≥ 0 → s(2)*s(1) = 2*P) ∧ (P < 0 → s(2)*s(1) = -2*P) }**
3: store 1 $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
4: goto 8 $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
// Add const 2 5 **{(P ≥ 0 → 2*(-v0) = 2*P) ∧ (P < 0 → 2*(-v0) = -2*P) }**
5: load 0 $\{(P \geq 0 \rightarrow s(1) * (-v0) = 2 * P) \wedge (P < 0 \rightarrow s(1) * (-v0) = -2 * P)\ \}$
6: neg $\{(P \geq 0 \rightarrow s(1) * (-s(0)) = 2 * P) \wedge (P < 0 \rightarrow s(1) * (-s(0)) = -2 * P)\ \}$
// Add mul 7 **{(P ≥ 0 → s(2)*s(1) = 2*P) ∧ (P < 0 → s(2)*s(1) = -2*P) }**
7: store 1 $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
8: load 1 $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
9: return $\{(P \geq 0 \rightarrow result = 2 * P) \wedge (P < 0 \rightarrow result = -2 * P)\ \}$

Figure 8: *WP calculus on an annotated bytecode*

REFERENCES

Freund, S. N and Mitchell, J. C, (1999) *A type system for object initialization in the Java bytecode language*. In ACM Trans. Program. Lang. Syst., vol 21, pp.1196–1250.

Grégoire, B, Sacchini, J. L and Sivan, R, (2008) *Combining a verification condition generator for a bytecode language with static analyses*. In Proceedings of the 3rd conference on Trustworthy global computing, Springer-Verlag, pp.23–40.

Binder, W and Hulaas, J, (2005) *Java Bytecode Transformations for Efficient, Portable CPU Accounting*. In Electron. Notes Theor. Comput. Sci., Elsevier Science Publishers B. V. vol 141, pp.53–73.

Noubissi,A.C, Iguchi-Cartigny, J and Lanet,J. L, (2011) *Hot updates for Java based smart cards*. In ICDE Workshops, pp.168-173.

Burdy, L and Pavlova,M, (2006) *Java bytecode specification and verification* In SAC 2006, pp.1835-1839.

Burdy,L, Huisman, M and Pavlova,M, (2007) *Preliminary Design of BML: A Behavioral Interface Specification Language for Java Bytecode* In FASE 2007, pp.215-229.

Freund, S. N and Mitchell, J. C,(2003) *A Type System for the Java Bytecode Language and Verifier*. In J. Autom. Reasoning, vol 30, pp.271-321.

Hoare,C. A. R, (1969) *An Axiomatic Basis for Computer Programming*. In Commun. ACM, vol 12, pp.576-580.

Stata, R and Abadi, M, (1999) *A Type System for Java Bytecode Subroutine* In ACM Trans. Program. Lang. Syst., vol21, pp.90-137.

Dahm,M, (1999) *Byte Code Engineering*. InJava-Informations-Tage, pp.267-277.

Sakamoto, T, Sekiguchi, T and Yonezawa, A, (2000) *Bytecode Transformation for Portable Thread Migration in Java*. In ASA/MA, 2000, pp.16-28.

Bachrach, J and Playford, K, (2001) *The Java Syntactic Extender*. In OOPSLA 2001, pp.31-42.

Noubissi,A. C, Iguchi-Cartigny, J and Lanet, J. L, (2010) *Incremental Dynamic Update for Java-Based Smart Cards*. In Fifth International Conference on Systems, pp.110-113.

Burdy,L, Cheon,Y, Cok, D. R, Ernst, M. D, Kiniry,J. R., Leavens,G. T, Leino, K. R. M, and Poll, E. *An Overview of JML Tools and Applications* . In Int. J. Softw. Tools Technol. Transf., vol 7, pp. 212–232.

Bannwart, F and Müller, P, (2005) *A Program Logic for Bytecode*. In Electron. Notes Theor. Comput. Sci.vol 141, Elsevier Science Publishers B. V., 2005, pp 255–273.

Common Criteria,http://www.commmoncriteria.org

McGachey, P, Hosking,A. L and Moss, J.E.B, (2009) *Pervasive Load-Time Transformation for Transparently Distributed Java*. In Electron. Notes Theor. Comput. Sci., vol 253, Elsevier Science Publishers B. V., pp.47–64.

Puder, P and Lee, J, (2009) *Towards an XML-based Bytecode Level Transformation Framework*. In Electron. Notes Theor. Comput. Sci.,vol 253, Elsevier Science Publishers B. V., pp.97–111.

Boshernitsan, M and Graham,S. L, (2004) *iXj: Interactive Source-to-source Transformations for Java*. In Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, pp.212–213.

Guodong, L, (2010) *Formal verification of programs and their transformations*. PhD thesis, University of Utah, USA.

Lounas,R, Mezghiche, M and Lanet,J. L, (2012) *Towards a General Framework for Formal Reasoning about Java Bytecode Transformation* In Proceedings Fourth International Symposium on Symbolic Computation in Software Science, pp.63–73.

Noubissi,A. C, (2011) *Mise á jour dynamique et sécurisée de composants systéme dans une carte á puce*. PhD thesis, University of Limoges, France, 2011.

Albert, E, Gomez-Zamalloa, M, Hubert, L and Puebla,G, (2007) *Verification of Java Bytecode Using Analysis and Transformation of Logic Programs* . In Practical Aspects of Declarative Languages,2007, Springer Berlin Heidelberg,pp.124-139.

Neamtiu,I, Hicks,M, Stoyle, G and Oriol, M, (2006) *Practical Dynamic Software Updating for C*.In ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 7283, 2006.

Gupta, D, Jalote P and Barua, G. *A formal framework for online software version change*. Software Engineering, IEEE Transactions on, 22 (2):120131, 1996.

Orso,A, Rao, A and Harrold,M. J. *A technique for dynamic updating of Java Software*. In ICSM, 2002.

Hlopko, M, Kurs,J, and Vrany, J. *Towards a Runtime Code Update in Java an exploration using STX:LIBJAVA*. In proceeding of Dateso 2013.

State Space Reduction Strategies for Model Checking Concurrent C Programs

Amira Methni, Belgacem Ben Hedia, Matthieu Lemerre,
CEA, LIST, Centre de Saclay,
PC172, 91191, Gif-sur-Yvette, FRANCE
{*amira.methni,belgacem.ben-hedia,
matthieu.lemerre*}*@cea.fr*

Serge Haddad
LSV, ENS Cachan, CNRS
& INRIA, France
haddad@lsv.ens-cachan.fr

Kamel Barkaoui
CEDRIC Lab, CNAM,
Paris, France
kamel.barkaoui@cnam.fr

Model checking is an effective technique for uncovering subtle errors in concurrent systems. Unfortunately, the state space explosion is the main bottleneck in model checking tools. Here we propose a state space reduction technique for model checking concurrent programs written in C. The reduction technique consists in an analysis phase, which defines an approximate *agglomeration predicate*. This latter states whether a statement can be agglomerated or not. We implement this predicate using a syntactic analysis, as well as a semantic analysis based on abstract interpretation. We show the usefulness of using agglomeration technique to reduce the state space, as well as to generate an abstract TLA+ specification from a C program.

Model checking, TLA, State space reduction, Agglomeration predicate.

1. INTRODUCTION

Model checking is an attractive formal verification technique because it is automatic. It offers extensive and thorough coverage by considering all possible behaviors of a system, unlike traditional testing methods. Given a set of properties expressed in a temporal logic and a model, model-checking automatically analyzes the state space of the model and checks whether the model satisfies the properties (Clarke et al. 1999). However, the main obstacle of model checking is the state explosion problem and concurrency is a major contributor to this problem.

Many solutions have already been investigated for reducing the complexity of model checking. For instance, by getting a simpler model from the original one using abstraction technique (Clarke et al. 1994), or by using on-the-fly model checking to eliminate part of the search to the automaton representing the (negation of the) checked property (Fernandez et al. 1992).

1.1. Contribution

In this paper, we present a state space reduction technique for model checking concurrent programs written in a low level language. We apply this technique to the verification of C programs by an explicit model checker. We use TLA+ (Lamport 1994) as a formal specification language for our concurrent C programs and we base ourselves on previous work reported in (Methni et al. 2015). The reduction technique is based on an analysis phase, which defines an approximate *agglomeration predicate*. This latter states whether a statement can be agglomerated or not. We implement this predicate using a syntactic analysis, as well as a semantic analysis based on abstract interpretation of C code. The particularity of our method is that we apply the reduction technique during the generation of TLA+ code and by using the abstract interpretation technique. We show the usefulness of using this technique to reduce the state space during the verification of C programs, as well as to generate an abstract TLA+ specification from a C program.

1.2. Outline

The rest of the paper is organized as follows. We give an overview of TLA+ in Section 2. Section 3 presents how we specify the semantics of C in TLA+. Section 4 describes the reduction technique and how we implement it on C programs. Experimental results are presented in Section 5. We discuss related work in Section 6. Section 7 concludes and illustrates future research directions.

2. OVERVIEW OF TLA

TLA+ (Lamport 2002) is the specification language of the Temporal Logic of Actions (TLA). TLA is

$\langle formula \rangle$	$\triangleq$	$\langle predicate \rangle \mid \Box[\langle action \rangle]_{\langle state\ function \rangle} \mid \neg\langle formula \rangle$ $\mid \langle formula \rangle \wedge \langle formula \rangle \mid \Box\langle formula \rangle$
$\langle action \rangle$	$\triangleq$	boolean valued expression containing constant symbols, variables, and primed variables
$\langle predicate \rangle$	$\triangleq$	$\langle formula \rangle$ with no primed variables | ENABLED $\langle action \rangle$
$\langle state\ function \rangle$	$\triangleq$	nonboolean expression containing constant symbols and variables

Figure 1: *TLA syntax (Lamport 2002)*

a variant of linear temporal logic introduced by (Lamport 1994) for specifying and reasoning about concurrent systems. Readers interested in a more detailed presentation of TLA+ can refer to Lamport's book (Lamport 2002).

TLA+ specifies a system by describing its possible behaviors. A *behavior* is an infinite sequence of states. A *state* is an assignment of values to variables. A *state function* is a nonboolean expression built from constants, variables and constant operators and it assigns a value to each state. For example, $y + 3$ is a state function that assigns to state s the value 3 plus the value that s assigns to the variable y. An *action* is a boolean expression containing constants, variables and primed variables (adorned with "′" operator). Unprimed variables refer to variable values in the current state and primed variables refer to their values in the next-state. Thus, an action represents a relation between old states and new states. A *state predicate* (or predicate for short) is an action with no primed variables.

The syntax of TLA is given in Figure 1 (the symbol $\triangleq$ means *equal by definition*). TLA+ formulas are built up from actions and predicates using boolean operators ($\neg$ and $\wedge$ and others that can be derived from these two), quantification over logical variables ($\forall, \exists$), and the unary temporal operator $\Box$ (*always*) of the linear temporal logic (Manna and Pnueli 1992).

The predicate "ENABLED $\mathcal{A}$", where $\mathcal{A}$ is an action, is defined to be true in a state s iff there exists some state t such that the pair of states $\langle s, t \rangle$ satisfies $\mathcal{A}$. The formula $[\mathcal{A}]_{vars}$, where $\mathcal{A}$ is an action and $vars$ the tuple of all system variables, is equal to $(\mathcal{A} \vee (vars' = vars))$ where $vars'$ is the expression obtained by priming all variables in $vars$. It asserts that every step (pair of successive states) is either an $\mathcal{A}$ step or else leaves the values of all variables $vars$ unchanged. TLA+ defines the abbreviation "UNCHANGED $vars$" to denote that $vars' = vars$.

While TLA+ permits a variety of specification styles, the specification that we use is defined by:

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge Fairness \qquad (1)$$

where:

- $Init$ is a state predicate describing the possible initial states by assigning values to all system variables,
- $Next$ is an action representing the program's next-state relation,
- $vars$ is the tuple of all variables,
- $Fairness$ is an optional formula representing weak or strong assumptions about the execution of actions.

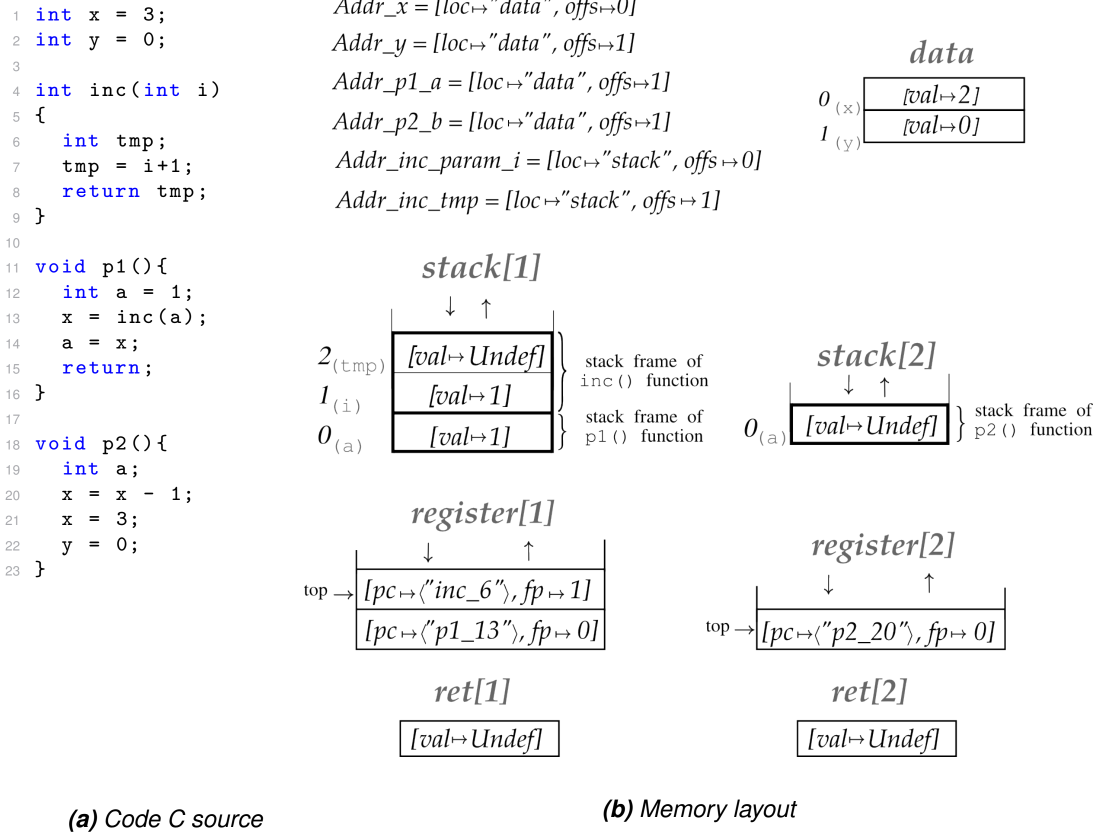
Formula $Spec$ is true of a behavior σ iff $Init$ is true of the first state of σ and every step of σ is either a $Next$ step or a "stuttering step", in which none of the specified variables change their values, and $Fairness$ holds. The behaviors satisfying the specification formula given by Equation (1) are the ones that represent correct behaviors of the system, where a behavior represents a conceivable history of a universe that may contain the system.

The TLA+ formula $Spec \Rightarrow \phi$ is valid when the model represented by $Spec$ satisfies the property ϕ, or implements the model ϕ.

TLA+ has an explicit model checker called TLC that can be used to check the validity of safety and liveness properties. TLC handles specifications that have the standard form of the formula (1). For this reason, we only use specification formula of the form of Equation (1). TLC requires a configuration file which defines the finite-state instance to analyze. It begins by generating all states satisfying the initial predicate $Init$. Then, it generates every possible next-state t such that the pair of states $\langle s, t \rangle$ satisfies $Next$ and the $Fairness$ constraints, looking for a state where an invariant is violated. Finally, it checks temporal properties over the state space.

3. TRANSLATION FROM C TO TLA+

Our approach to checking a concurrent C program is to first translate it into a TLA+ specification, to which the TLA+ tools can be applied. In what follows, we briefly present how we specify the semantics of C in



(a) *Code C source* ***(b)*** *Memory layout*

Figure 2: *Example of a C code in which one process (with id equals to 1) executes `p1()` function and the second one executes `p2()`. The top of the $stack[1]$ indicates that process 1 is executing the statement with label 6 of `inc()` function. $Undef$ represents an undefined value such as the value of an uninitialized variable.*

TLA+ by describing the memory layout considered and how we model the control flow of a C program.

3.1. Memory Layout

C file is parsed and normalized according to CIL (C Intermediate Language) (Necula et al. 2002) which transforms complicated constructs of C into simpler ones. This transformation makes programs more amenable to analysis and transformation. According to the Abstract Syntax Tree (AST) of the C program, C2TLA+ generates automatically a TLA+ specification according to a set of translation rules detailed in our previous work (Methni et al. 2015).

In C2TLA+, a concurrent program consists of many interleaved sequences of operations called *processes*, corresponding to threads in C. Each process has a unique identifier id. The set of all processes is determined by the TLA+ constant $ProcSet$.

Figure 2 presents a C program and the content of the memory as modeled by C2TLA+. We consider that the C code is executed by two processes. One process executes `p1()` function and the other one executes `p2()` function.

The memory is separated into four areas that do not overlap:

- a shared memory called $data$ that stores global (and static) variables. In the example of Figure 2a, the `x` variable is shared by the two processes.
- a local memory for each process, called $stack$ and stores local variables and function parameters. The memory $stack[id]$ specifies the local memory of process id and is composed of stack frames. Each stack frame corresponds to a call to a function. In the example of Figure 2a, $stack[1]$ is composed of two stack frames, one of `p1()` function and one of `inc()` function. When a function call terminates, its stack frame is removed.
- a local memory for each process called $register$ modeled as a sequence and stores the program counter of each process. The head of this sequence contains the statement being currently executed by the process id.
- a local memory called ret which contains values to be returned by processes.

The memory is modeled in TLA+ by a variable called $memory$. It is a record whose four fields represent the four memory areas. The global memory $data$ behaves like an array of values, whereas $stack$ and $register$ behave like a FIFO (First In, First Out) queues. Access to those memory areas is addressed using offsets. So, a memory address is a couple $[loc, offs]$ of memory location loc, ($data$ or $stack$ area) and an offset $offs$ in this location. For instance, $Addr_x$ specified in Figure 2b defines the memory address of `x` variable.

The main operations that manage the memory are $load()$ and $store()$:

- $load()$ is the function that given the current state of memory mem and a memory address $addr$ (in the form of $[loc, offs]$), returns the value stored at the address $addr$ in the memory mem,
- $store()$ is the function that given the current state of memory mem, a memory address $addr$ and a value val, returns the new copy of the memory after storing the value val at the memory address $addr$.

3.2. Specifying the C control-flow

Each C statement i is identified in C2TLA+ by a label assigned by CIL and is modeled by a TLA+ function, noted $stmt_i()$, which takes as arguments the process identifier id and the memory mem, and returns the new content of the memory after executing the statement.

Each $stmt_i()$ updates the program counter $register$ of the process id and may change the content of mem, $stack$, and/or ret memory areas depending on the type of the statement (assignment, jump statements, etc.). For instance, the statement on line 20 is translated into the TLA+ function $p2_20(id, mem)$ defined as follows:

$$
\begin{array}{l}
p2_20(id, mem) \triangleq \\
\quad \text{LET } mcopy \triangleq load(id, Addr_x, [val \mapsto 3]) \text{ IN} \\
\quad\; [data \mapsto mcopy.mem, stack \to mcopy.stack, \\
\quad\; register \mapsto [mem.register \text{ EXCEPT } ![id] = \\
\quad\; \langle [pc \mapsto \langle \text{“}p2_21\text{”} \rangle, fp \mapsto Head(mem.register[id]).fp] \rangle \\
\qquad\quad \circ\, Tail(mem.register[id]), \\
\quad\; ret \mapsto mem.ret]
\end{array}
$$

The definition of $p2_20()$ function uses the TLA+ construct LET/IN to define a temporary variable that stores the value of the memory after affecting the value 3 to the memory address $Addr_x$. The symbol $\circ$ defines the concatenation operator for TLA+ sequences. $Head(s)$ is a TLA+ function that returns the head of the sequence s and $Tail(s)$ returns the tail of the sequence s. Then, the $register[id]$ is updated by the label value of the successor statement given by the control flow graph (CFG) of the C program (provided by CIL).

The control flow of the C program in C2TLA+ is ensured by the $dispatch()$ function. For the example of Figure 2a, this function is defined as follows:

$$
\begin{array}{l}
dispatch(id, mem) \triangleq \\
\quad \text{CASE } Head(mem.register[id]).pc = \text{”}inc_6\text{”} \\
\qquad\quad \to inc_6(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}inc_7\text{”} \\
\qquad\quad \to inc_7(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p1_11\text{”} \\
\qquad\quad \to p1_11(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p1_12\text{”} \\
\qquad\quad \to p1_12(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p1_13\text{”} \\
\qquad\quad \to p1_13(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p1_14\text{”} \\
\qquad\quad \to p1_14(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p2_19\text{”} \\
\qquad\quad \to p2_19(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p2_20\text{”} \\
\qquad\quad \to p2_20(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p2_21\text{”} \\
\qquad\quad \to p2_21(id, mem) \\
\qquad \Box\; \text{OTHER } \to mem
\end{array}
$$

The $dispatch()$ function calls, according to the value of the pc field contained at the top the process register (determined by the expression $Head(mem.register[id]).pc$), the corresponding TLA+ function to execute, i. e., the C instruction to execute.

The C program is thus simulated by the $Spec$ formula given by equation (1). The $Init$ predicate specifies the initial values of the memory and the $Next$ action is defined as follows:

$$
\begin{array}{l}
Next \triangleq \\
\quad \lor\; \exists\, id \in ProcSet : \\
\qquad\quad \land\; memory.regsiter[id] \neq \langle\rangle \\
\qquad\quad \land\; memory' = dispatch(id, mem) \\
\quad \lor\; \forall\, id \in ProcSet : \\
\qquad\quad \land\; memory.regsiter[id] = \langle\rangle \\
\qquad\quad \land\; \text{UNCHANGED } memory
\end{array}
$$

It states that one of the processes that has not finished execution (its $register[id]$ is not empty) is nondeterministically chosen to execute one action until all processes finish execution, i. e., all registers become empty. Executing an action consists in calling $dispatch()$ function. For example, when $Head(mem.register[id]).pc$ equals to ”inc_6”, calling the function $inc_6(id, mem)$ will update the value of $stack[id]$ (as `tmp` is stored in the local memory) as well as the top of $register[id]$. As the $register[id]$ is

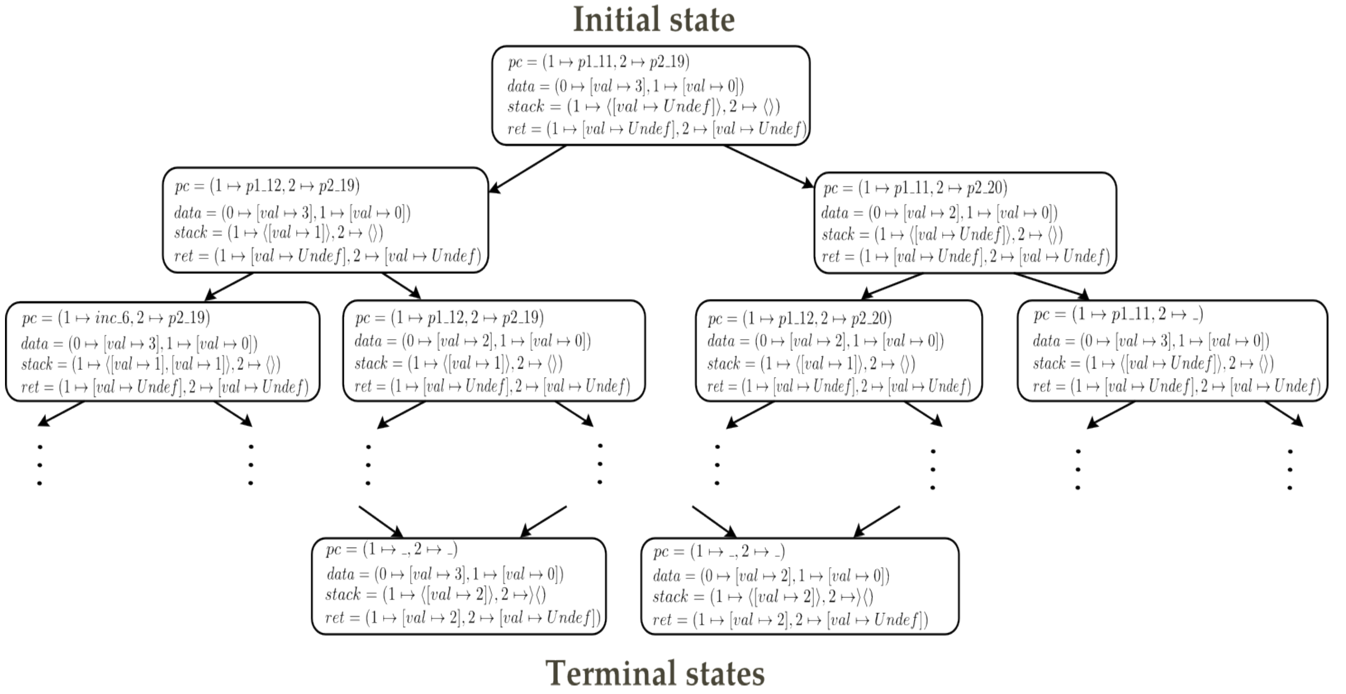


Figure 3: *Example of a state space*

still not empty, the control flow is thus passed to the successor statement.

The behavior of the C program modeled by the $Spec$ formula can be given in terms of a *state transition system*.

Definition 1. A state transition system is a 3-tuple $T = (Q, I, \delta)$ given by

- a finite set of states Q,
- a set $I \subseteq Q$ of initial states, specified by the $Init$ predicate,
- a transition relation $\delta \subseteq Q \times Q$ that links two states. This latter corresponds to satisfying the predicate $Next$.

The state transition system encodes the state space of the corresponding TLA+ specification of the C program.

Figure 3 illustrates the state space of the corresponding TLA+ specification of the C code given in Figure 2a. It consists in all the possible interleaving of process execution. In order to simplify, we represent only the content of pc field contained at the top of the $register[1]$ and $register[2]$ memories. Each state of the graph matches a valuation of $memory$ variable, i. e., its four fields.

3.3. Process Synchronization

All processes interact with each other through the shared memory $data$. Concurrent access to this latter is ensured via synchronization mechanism. There are many different ways to implement concurrency synchronization in C. For instance, by using locks and semaphores, or by providing low level hardware instructions (e. g., *test-and-set* and *compare-and-swap*). To support synchronization mechanism, generated TLA+ specifications by C2TLA+ can be completed with manually written TLA+ specifications to provide concurrency primitives and atomic instructions. More detailed information about integrating synchronization primitives in TLA+ specifications can be found in our previous work (Methni et al. 2015).

4. APPLYING REDUCTION ON C PROGRAMS

The process of generating an optimized TLA+ specification is illustrated in Figure 4. To apply reduction on C programs, it is necessary to define the agglomeration predicate. The C program is first analyzed. This analysis phase defines an approximate *agglomeration predicate* which takes as argument a C statement and returns true or false depending on whether the statement can be agglomerated or not. This predicate can be *safe* or *unsafe*. The meaning of *safe predicate* depends on how the analysis is performed.

- The predicate is said safe when the analysis is a safe approximation. Its definition is as follows:
  - if the statement does not modify the shared memory, the predicate returns true,

  - if the statement modifies the shared memory or it is *unknown*, it returns false.

  The *unknown* predicate states that we have no idea if the statement modifies the memory or not.

- The predicate is called unsafe when a statement is agglomerated and we are not sure if it modifies the global memory or not.

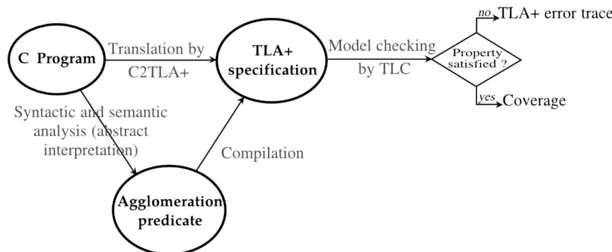


Figure 4: *Reduction process*

In this Section, we introduce the agglomeration technique by an example. Then, we describe the implementation of a safe agglomeration predicate by using a syntactic and semantic analysis on C programs. Then, we show the interest of using an unsafe agglomeration predicate to generate an abstract TLA+ specification of a C program.

In what follows, we use the expression agglomerating TLA+ actions, to designate agglomerating the corresponding statements in the C program.

4.1. An introducing example

As the semantics of a TLA+ is expressed through a state transition system, where transitions between states are ensured by TLA+ actions, the reduction technique consists in *agglomerating* consecutive actions into one atomic action which performs the effects of the original ones. The reduction idea based on agglomeration has been widely used in Petri Nets (Haddad and Pradat-Peyre 2006; Berthelot 1986).

Figure 5 shows three consecutive states linked by two actions $x' = x + 1$ and $x' = x + 2$. The result of this agglomeration (represented by $\longrightarrow$), is two states linked by one atomic action which is the result of executing the action $x' = x + 1$, then $x' = x + 2$.

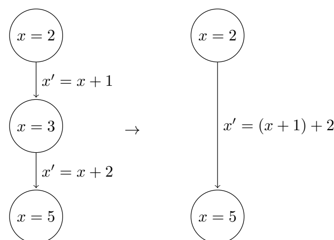
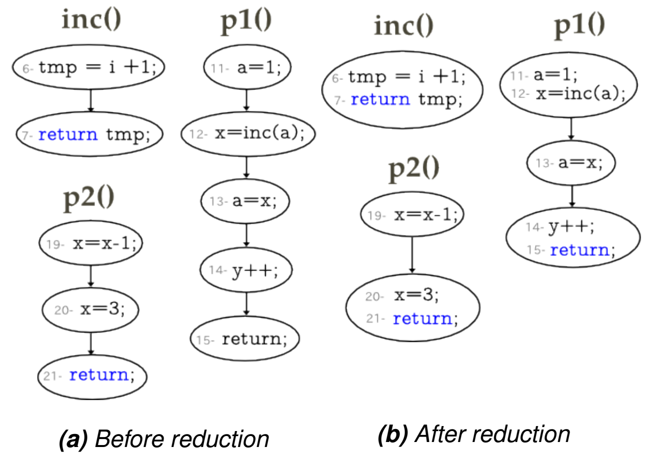


Figure 5: *Agglomerating actions*



(a) *Before reduction* ***(b)*** *After reduction*

Figure 6: *The control flow graph of a C code example*

4.2. Using syntactic and semantic analysis

Syntactic analysis
A syntactic analysis is performed to detect statements on which reduction can be applied. Often, C functions make use of local variables, and when a statement refers only to local variables, the value for which the statement is executed by a process cannot change the execution of other processes. Furthermore, we assume that statements involving local pointer variable cannot be agglomerated as they may reference shared memory.

Moreover, we consider that jump statements, namely `goto`, `break` and `continue`, can be agglomerated with its successor (designated by computing the CFG), as they only change the local register of the process ($register[id]$ in TLA+ specification).

Semantic analysis
In many scenarios, a concurrent C program could contain, in its global memory, data blocks that are accessed only by one process at a time. In that case, syntactic analysis is insufficient. Therefore, we use a semantic approximation predicate. The C program is thus analyzed and an approximation of memory access is computed using the Mthread Frama-C plugin (Mth). This latter provides information about the memory zones that are accessed concurrently by more than one process and those that are not. In this case, the agglomeration predicate is *safe* as the analysis is based on an over-approximation of the memory.

We consider the example given by Figure 2a. To illustrate the agglomeration technique on this example, we represent the C program by its control flow graph, illustrated by Figure 6a, where each state of the graph represents a C statement and edges represent the control flow. After applying the syntactic and semantic analysis on this example,

the control flow graph is transformed into a smaller graph, given in Figure 6b. Each state of the graph corresponds to one statement or a block of statements. As the `inc()` function uses only local memory, its block definition is combined into one basic block. For `p1()` function, statement on line 11 is agglomerated with statement on line 12 and statement on line 14 is agglomerated with the `return` instruction.

4.3. Generating an abstract specification

Agglomerating statements can also be useful to generate an abstract TLA+ specification of a C program. The user can define which C statements can be agglomerated using an unsafe predicate. The resulting TLA+ specification can be viewed as an abstract formal specification of the C program.

As TLA+ is a fragment of $\mathrm{LTL}_{\backslash x}$ (Linear Temporal Logic without the “next operator”), it is well known that the equivalence between checking a property given in $\mathrm{LTL}_{\backslash x}$ on an abstract model and checking it on the original model is ensured by the preservation (Goltz et al. 1992).

Example: Agglomerating critical sections
Consider the following fragment of C code (Figure 7) implementing an example of the producer/consumer model. The two processes share a buffer protected by a mutex `m`. The synchronization between processes is ensured by two semaphores `empty` and `full`. Mutex and semaphores are implemented as an integer values and are only accessible through two atomic operations `P()` and `V()`.

Translating this implementation into a TLA+ specification and model checking results in verifying all interleavings of actions between processes. We define the agglomeration predicate that states that statements protected by mutex (namely by `P()` and `V()` primitives) can be agglomerated. After reduction, the control flow graph of this example is illustrated in Figure 8b.

Therefore, the block statements from line 12 to line 14 and that from line 12 to 26 are agglomerated into one state. The state space of the TLA+ specification generated after agglomerations contains fewer states than the one without agglomerations as the reduction inside the critical section restricts the amount of interleaving allowed between processes. We define the mutual exclusion property in TLA+ as follows:

$$
\begin{aligned}
& mut_exclusion(lbl1, lbl2) \triangleq \Box((\forall\, id1, id2 \in ProcSet : \\
& \quad \land (id1 \neq id2) \land (Head(memory.register[id1]) \neq \langle\rangle) \\
& \quad \land (Head(memory.register[id2]) \neq \langle\rangle) \\
& \quad \land (Head(memory.regsiter[id1]).pc = lbl1)) \\
& \qquad \Rightarrow Head(memory.register[id2]).pc \neq lbl2)
\end{aligned}
$$

```
1  #define BUFFER_SIZE 5
2  mutex m;
3  sem full = 0, empty = BUFFER_SIZE;
4  int buffer[BUFFER_SIZE]; /* the buffer */
5  int count; /* buffer count */
6
7  void Producer(int item) {
8   while(TRUE) {
9    item = rand();/*generate a random number*/
10   P(&empty); /*acquire the empty lock*/
11   P(&m); /*acquire the mutex lock*/
12   if(count < BUFFER_SIZE) {
13     buffer[count] = item;
14     count++; }
15   V(&m); /*release the mutex lock*/
16   V(&full); /*signal full*/
17  }
18 }
19 void Consumer(void) {
20  while(TRUE) {
21   int item;
22   P(&full); /*acquire the full lock*/
23   P(&m); /*aquire the mutex lock*/
24   if(count > 0) {
25     item = buffer[(count-1)];
26     count--; }
27   V(&m); /*release the mutex lock*/
28   V(&empty); /*signal empty*/
29  }
30 }
```

Figure 7: *Example of a producer/consumer model using locks*

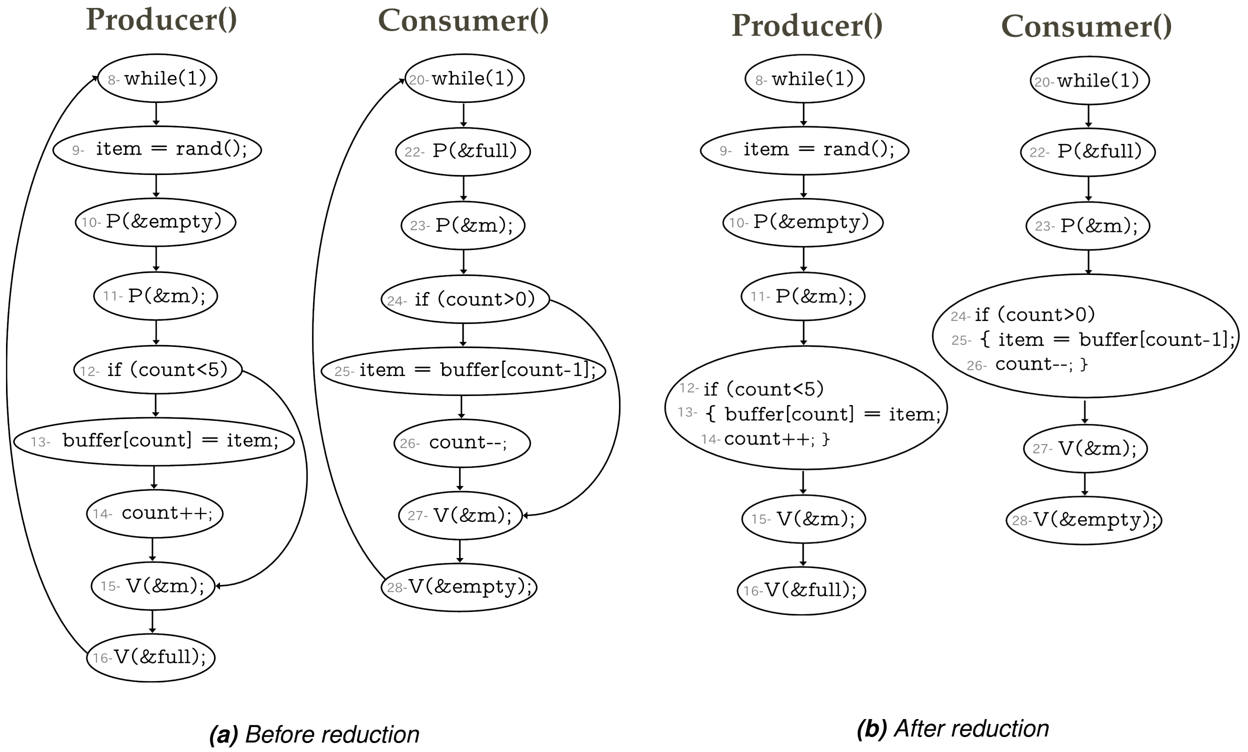
This property expresses that critical sections cannot be executed simultaneously. This property was verified on the TLA+ specification after reduction. Thus, we can deduce that the property is also verified on the specification generated without the reduction technique.

4.4. Integrating the reduction into TLA+ specification

In what follows, we show how we implement the reduction on TLA+ specification. As described in Section 3, each execution of $Next$ action corresponds to executing an atomic C statement. The reduction in C programs, consists in translating a sequence of C statements into one action instead of multiple ones. Let i be the identifier of a statement and j be the identifier of its successor. To do that, we generate for each statement i a new function that we call $stmt_long_i()$ defined below.

$$
\begin{aligned}
stmt_long_i(id, mem) \triangleq& \\
& stmt_long_j(id, dispatch(id, mem))
\end{aligned}
$$

The definition of $stmt_long_i(id, mem)$ consists in calling the function of the successor statement j, noted by $stmt_long_j()$ and passing as argument the memory state returned by $dispatch(id, mem)$.



(a) *Before reduction*

(b) *After reduction*

Figure 8: *The control flow graph of the* `Producer()` *and* `Consumer()` *functions before and after reduction*

To generate a reduced TLA+ specification, we iterate over all statements and when the agglomeration predicate returns true for a statement i, its translation consists in calling the function $stmt_long_j(id, dispatch(id, mem))$. Otherwise, we call $dispatch(id, mem)$ function.

The C program is thus specified by the formula $Spec$ given by equation (1), except that the $Next$ action calls a new function $dispatch_red()$, instead of $dispatch()$ function. For the example of figure 2a, the $dispatch_red()$ function is defined in Figure 9:

$$
\begin{aligned}
& dispatch_red(id, mem) \triangleq \\
& \quad \text{CASE } Head(mem.st[id]).pc = \text{"}inc_6\text{"} \\
& \qquad\qquad \rightarrow inc_long_6(id, mem) \\
& \quad \Box\ Head(mem.register[id]).pc = \text{"}inc_7\text{"} \\
& \qquad\qquad \rightarrow inc_long_7(id, mem) \\
& \quad \Box\ Head(mem.register[id]).pc = \text{"}p1_11\text{"} \\
& \qquad\qquad \rightarrow p1_long_11(id, mem) \\
& \quad \Box\ Head(mem.register[id]).pc = \text{"}p1_12\text{"} \\
& \qquad\qquad \rightarrow p1_long_12(id, mem) \\
& \quad \Box\ Head(mem.register[id]).pc = \text{"}p1_13\text{"} \\
& \qquad\qquad \rightarrow p1_long_13(id, mem) \\
& \quad \Box\ Head(mem.register[id]).pc = \text{"}p1_14\text{"} \\
& \qquad\qquad \rightarrow p1_long_14(id, mem) \\
& \quad \Box\ Head(mem.register[id]).pc = \text{"}p2_19\text{"} \\
& \qquad\qquad \rightarrow p2_long_19(id, mem) \\
& \quad \Box\ Head(mem.register[id]).pc = \text{"}p2_20\text{"} \\
& \qquad\qquad \rightarrow p2_long_20(id, mem) \\
& \quad \Box\ Head(mem.register[id]).pc = \text{"}p2_21\text{"} \\
& \qquad\qquad \rightarrow p2_long_21(id, mem) \\
& \quad \Box\ \text{OTHER} \rightarrow mem
\end{aligned}
$$

Figure 9: *Example of the $dispatch_red()$ function definition*

The $dispatch_red(id, mem)$ function calls according to the value the program counter pc contained at the head of $mem.register[id]$ the corresponding TLA+ function to execute.

5. EXPERIMENTS

The reduction technique is totally automatic and was integrated in C2TLA+ which is a Frama-C (Cuoq et al. 2012) plugin, implemented in OCaml. This Section is concerned with our practical experience. We use the Mthread plugin results and the syntactic analysis as described in Section 4 to implement our agglomeration technique.

We consider one sequential C program and four concurrent programs:

Table 1: *Comparing Model Checking Results with & without Reduction (time in seconds)*

Prorgam	#Proc	Without reduction		With reduction		Factor
		#St	#T	#St	#T	
Zunebug	1	389	0.147	2	0.136	99.48
Dekker	2	173	0.128	70	0.109	59.53
Peterson	2	107	1.37	22	0.131	79.43
	4	1.080.161	59.2	31.221	4.82	97.10
Bakery	2	2.389	1.91	223	1.67	90.66
	4	50.515.927	1560	835.355	76.6	98.36
Philos	4	9.791.509	366	146.106	12	98.5
	5	>619.309.984	25340	4.179.520	352	99.32

- Zunebug which is a bug in the internal clock driver of Zune 30GB music player. The source code is taken from (Weimer et al. 2010).
- Lamport's Bakery and Peterson algorithms obtained from (Raynal 2013) and Dekker mutual exclusion algorithm presented in (Dijkstra 1968).
- Dining philosopher problem. We use the solution that appears in Tanenbaum's book (Tanenbaum 2007).

These programs make typical examples for demonstrating the strength of the state space reduction. C2LTA+ takes as input a C program and generates for each one the corresponding TLA+ specification.

Using the TLC model checker, we compute the total number of generated states and we verify a set of properties on the two specifications.

Results of experiments are shown in Table 1, where #Proc denotes the process number, #St denotes the numbers of states and #T denotes the time for model checking in seconds. Columns 3 to 6 give information about the state space generated with and without applying the reduction technique. The last column indicates the reduction factor, the ratio between the state space generated without reduction and the one after applying the reduction technique.

All experimental results were performed on an Intel Core Pentium i7-2760QM machine with 8 cores (2.40GHz each), with 8Gb of RAM memory. For zunebug, one property to verify is program termination, which is a liveness property that we express as follows:

$$termination \triangleq \diamond(Head(memory.register[1]) = \langle\rangle)$$

This property asserts that the register of the program will eventually be empty. For the TLA+ specification without agglomeration, checking this property causes TLC to report an error. This error occurs when the code takes as input the last day of a leap year, causing the code to enter into an infinite loop. After applying the reduction technique for the zunebug program, the state space size of its corresponding TLA+ specification equals 2. This is due to the fact that the program is sequential. Model checking the TLA+ code with the last day of a leap year causes the TLC model to report an incorrect recursive function definition.

For the concurrent programs, the mutual exclusion property has been successfully verified on Peterson, Bakery, Dekker and Philosophers benchmarks. As expected, the size of the state space with agglomerations is always smaller than the one without agglomerations. For the philosopher example with 5 processes, the state space without agglomeration takes more than 7 hours to be model checked. However, using the reduction technique the specification is verified in 6 minutes. The reduction factor in this cas reaches 99.32. The reduction technique obtains good results on these benchmarks due to the elimination of some intermediate states.

6. RELATED WORK

There are a wealth of research contributions on formal verification of software as well as techniques for the reduction of the state space.

Program slicing is a technique introduced by (Weiser 1981) for simplifying sequential programs for debugging and program understanding. It consists in removing from the program features that are irrelevant for the property to be verified. Recently, slicing technique has been used to reduce the state space of a system in model checking. It has been applied to Promela (Millett and Teitelbaum 2000), the input language for the Spin model checker

(Holzmann 1997). The interested reader can refer to (Tip 1995) for a detailed description of the different approaches used in the program slicing.

Predicate abstraction (Graf and Saïdi 1997) is a technique in which a set of predicates over the programs variables is used to construct an abstract program. This technique is being used in SLAM (Ball and Rajamani 2002), BLAST (Henzinger et al. 2003) and MAGIC (Chaki et al. 2004).

Other approaches perform reduction during exploration of the state space of the program. For example, partial order reduction (Valmari 1989) is a technique which explores only a representative subset of the state space of a model. The basic idea is to exploit the commutativity caused by the interleavings of transitions, which result in the same state. This technique was first introduced for checking the absence of deadlock. Subsequently, a number of variants of this technique have been developed and integrated in verification tools, like Spin (Holzmann 1997) and Verisoft (Godefroid 1997).

Although we have mentioned some projects in the C context, there are also significant works interested in model checking the Java language. For example, JPF (Visser et al. 2003) uses state compression technique to handle big states, partial order and symmetry reduction, slicing, abstraction and runtime analysis techniques to reduce the state space.

In this work, the state space reduction technique that we propose is closer to that originally introduced by (Berthelot 1986) in Petri nets formalism. Berthelot developed a large set of reduction rules for reducing the complexity of verification. Extended work has been proposed by (Haddad and Pradat-Peyre 2006). Our work differs from this latter by the fact that the model of our TLA+ specification is a state transition system and the agglomeration predicate depends on the analysis of the C program. Our reduction technique is applied during the generation of TLA+ code unlike the partial order reduction technique which performs reduction during the construction of the state space. Besides, we use TLA+ as formal framework which provides an expressive power to specify the semantics of a programming language and can reason about concurrent systems and can express safety and liveness properties unlike SLAM and BLAST which have limited support for concurrent properties as they only check safety properties.

7. CONCLUSION AND FUTURE WORK

We have proposed a technique to reduce the state space for model checking C programs. We used C2TLA+ to translate the semantics of C to the formal specification language TLA+. This reduction technique is based on an analysis phase, which defines an approximate *agglomeration predicate* that states whether a statement can be agglomerated or not. We implemented this predicate by applying a syntactic and semantic analysis on C Programs. We illustrated the effectiveness of applying the agglomeration technique to reduce the state space during the verification of C programs and also as well as to define an abstract TLA+ specification that model the behavior of C programs.

We aim to integrate a mechanism for structuring large TLA+ specifications from C programs using a refinement process between different levels of abstraction. Finally, we are planning to apply the methodology on a critical part of the microkernel of the PharOS (Lemerre et al. 2011) real-time operating system (RTOS).

REFERENCES

Mthread plugin. URL `http://frama-c.com/mthread.html`.

Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging System Software via Static Analysis. *SIGPLAN Not*, 2002.

Gérard Berthelot. Checking properties of nets using transformation. In *Advances in Petri Nets 1985, Covers the 6th European Workshop on Applications and Theory in Petri Nets-selected Papers*, pages 19–40, London, UK, UK, 1986. Springer-Verlag.

Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag. URL `http://frama-c.com/`.

Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

Jean-Claude Fernandez, Laurent Mounier, Claude Jard, and Thierry Jron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1(2-3):251–273, 1992.

Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *In Proceeedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186. ACM Press, 1997.

Ursula Goltz, Ruurd Kuiper, and Wojciech Penczek. Propositional Temporal Logics and Equivalences. In W.R. Cleaveland, editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 222–236. Springer Berlin Heidelberg, 1992.

Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, UK, 1997. Springer-Verlag.

Serge Haddad and Jean-François Pradat-Peyre. New Efficient Petri Nets Reductions for Parallel Programs Verification. *Parallel Processing Letters*, 16(1):101–116, 2006.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software Verification with BLAST. pages 235–239. Springer, 2003.

Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

Leslie Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

M. Lemerre, E. Ohayon, D. Chabrol, M. Jan, and M-B. Jacques. Method and Tools for Mixed-Criticality Real-Time Applications within PharOS. In *Proceedings of AMICS 2011: 1st International Workshop on Architectures and Applications for Mixed-Criticality Systems*, 2011.

Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Serge Haddad, and Kamel Barkaoui. Specifying and Verifying Concurrent C Programs with TLA+. In *Formal Techniques for Safety-Critical Systems*, volume 476 of *Communications in Computer and Information Science*, pages 206–222. Springer, 2015.

Lynette I. Millett and Tim Teitelbaum. Issues in Slicing PROMELA and its Applications to Model Checking, Protocol Understanding, and Simulation. *International Journal on Software Tools for Technology Transfer*, 2(4):343–349, 2000.

George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, Heidelberg, 2013.

Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

Antti Valmari. Stubborn Sets for Reduced State Space Generation. In *Proceedings of the Tenth International Conference on Application and Theory of Petri Nets*, pages 1–22, 1989.

Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engg.*, 10(2):203–232, April 2003.

Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic Program Repair with Evolutionary Computation. *Commun. ACM*, 53(5):109–116, May 2010.

Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

Part III

Session: Performance evaluation

Timeout Interaction and Migration in Distributed Systems

Gabriel Ciobanu
Romanian Academy, Institute of Computer Science, Iaşi, Romania
gabriel@info.uaic.ro

The complexity of distributed systems is increasing, and so they require appropriate formalisms and techniques for their specificatio and verification Since these distributed systems grow more complex and more powerful, it is important to fin scaling formal methods for both specificatio and verifi cation. Successful formalisms for specificatio and verificatio of certain distributed systems are given by networks of timed automata and by Petri nets; however, these formalisms are not easily scalable, a reason why we look for compositional specificatio and verificatio techniques. In terms of specification a process calculus would solve the compositional issue. Moreover, in distributed systems coordination is given by time scheduling, access to resources, and interaction among processes. When modelling distributed systems it is useful to have an explicit notion of location, local clocks, explicit migration and resource management.

We have introduced in Ciobanu and Koutny (2008) a rather simple and expressive formalism called TiMo as a simplifie version of timed distributed π-calculus Ciobanu and Prisacariu (2006) which is an extension of distributed π-calculus Hennessy (2007). TiMo is a process calculus with explicit migration allowing the use of timers for controlling process mobility and interaction. Migration involves several explicit locations. Each location has a local clock, modelling distributed systems in a more accurate way. Timing constraints for migration allow to specify a temporal timeout after which a process must move to another location. Two processes may communicate only if they are present at the same location. A timer denoted by $\Delta 3$ associated to a migration action $\mathrm{go}^{\Delta 3} work$ indicates that the process moves to location $work$ after at most 3 time units. It is also possible to indicate a deadline for a communication over a channel; if a communication action does not happen before this deadline, the process gives up and switches its operation to an alternative process. E.g., a timer $\Delta 5$ associated to an output action $a^{\Delta 5}!\langle v \rangle$ makes the channel available for communication only for a period of 5 time units. Considering suitable data sets including a set *Loc* of locations, a set *Chan* of communication channels and a set *Id* of process identifier , the syntax of TiMo is presented in Table 1.

Using TiMo , we can specify and analyse complex timing systems in a new and intuitive way. Aiming to bridge the gap between the existing theoretical approach of process calculi and forthcoming realistic programming languages for distributed systems, TiMo represents in several aspects a prototyping language for multi-agent systems featuring mobility and local interaction. Starting with a firs version proposed in Ciobanu and Koutny (2008), several variants were developed during the last years. We mention here the access permissions given by a type system in perTiMo Ciobanu and Koutny (2011a), as well as a probabilistic extension pTiMo Ciobanu and Rotaru (2013). Inspired by TiMo , a fl xible software platform was introduced in Ciobanu and Juravle (2009, 2012) to support the specificatio of agents allowing timed migration in a distributed environment.

In terms of verification interesting properties described by TiMo regarding could be analysed and checked. The properties of distributed systems described by TiMo refer to process migration, time constraints, bounded liveness and optimal reachability Aman et. all (2012); Ciobanu and Koutny (2011b). A verificatio tool called TiMo@PAT Ciobanu and Zheng (2013) was developed by using Process Analysis Toolkit (PAT), an extensible platform for model checkers. A formal relationship between

P	$::=$	$a^{\Delta lt}!\langle\vec{v}\rangle$ then P else P' |	(output)
		$a^{\Delta lt}?(\vec{u}{:}\vec{X})$ then P else P' |	(input)
		go$^{\Delta lt}$ l then P |	(move)
		$P \mid P'$ |	(parallel)
		0 |	(termination)
		$id(\vec{v})$	(definition
		ⓈP	(stalling)
L	$::=$	$l[\![P]\!]$	Located Processes
N	$::=$	L | $L \mid N$	Networks

Table 1: *TiMo Syntax.*

rTiMo and timed automata presented in Aman and Ciobanu (2013) allows the use of model checking capabilities provided by the well-known verificatio tool UPPAAL . A probabilistic temporal logic called PLTM was introduced in Ciobanu and Rotaru (2013) to verify complex probabilistic properties making explicit reference to specifi locations, temporal constraints over local clocks and multisets of actions.

Acknowledgements. The work was supported by a grant of the Romanian National Authority for Scientifi Research, project PN-II-ID-PCE-2011-3-0919.

REFERENCES

Aman, B. and Ciobanu, G. (2013) Real-Time Migration Properties of rTiMo Verifie in UPPAAL . In Hierons, R., Merayo, M.and Bravetti, M. (Eds.), SEFM 2013. *Lecture Notes in Computer Science* **8137**, 31–45.

Aman, B., Ciobanu, G. and Koutny, M. (2012) Behavioural Equivalences over Migrating Processes with Timers. In Giese, H. and Rosu, G. (Eds.) FMOODS/FORTE 2012, *Lecture Notes in Computer Science* **7273**, 52–66.

Ciobanu, G. (2008) Behaviour Equivalences in Timed Distributed π-Calculus. In Wirsing, M., Banâtre, J.-P., Hölzl, M. and Rauschmayer, A. (Eds.), *Lecture Notes in Computer Science* **5380**, 190–208.

Ciobanu, G. and Juravle, C. (2009) A Software Platform for Timed Mobility and Timed Interaction. In Lee, D., Lopes, A. and Poetzsch-Heffter, A. (Eds.) FMOODS/FORTE 2009, *Lecture Notes in Computer Science* **5522**, 106–121.

Ciobanu, G. and Juravle, C. (2012) Flexible Software Architecture and Language for Mobile Agents. *Concurrency and Computation: Practice and Experience***24**, 559–571.

Ciobanu, G. and Koutny, M. (2008) Modelling and Verificatio of Timed Interaction and Migration. In Fiadeiro, J.L., Inverardi, P. (Eds.) FASE 2008, *Lecture Notes in Computer Science* **4961**, 215–229.

Ciobanu, G. and Koutny, M. (2011) Timed Migration and Interaction With Access Permissions. In Butler, M., Schulte, W. (eds.) FM 2011, *Lecture Notes in Computer Science* **6664**, 293–307.

Ciobanu, G. and Koutny, M. (2011) Timed Mobility in Process Algebra and Petri nets. *The Journal of Logic and Algebraic Programming* **80(7)**, 377–391.

Ciobanu, G. and Prisacariu, C. (2006) Timers for Distributed Systems. In Di Pierro, A. and Wiklicky, H. (Eds.) QAPL 2006,*Electronic Notes in Theoretic Computer Science* **164(3)**, 81–99.

Ciobanu, G. and Rotaru, A. (2013) A Probabilistic Logic for pTiMo. In Liu, Z., Woodcock, J. and Zhu, H. (Eds.) ICTAC 2013, *Lecture Notes in Computer Science* **8049**, 141–158.

Ciobanu, G. and Zheng, M. (2013) Automatic Analysis of TiMo Systems in PAT. In *Proc. 18th International Conference on Engineering of Complex Computer Systems (ICECCS 2013)*, IEEE Computer Society, 121–124.

Hennessy, M. (2007) *A distributed π-calculus*. Cambridge University Press.

Model-Based Verification of the DMAMAC Protocol for Real-time Process Control

Admar Ajith Kumar Somappa
Bergen University College
University of Agder
aaks@hib.no

Andreas Prinz
University of Agder
andreas.prinz@uia.no

Lars M Kristensen
Bergen University College
lmkr@hib.no

Medium Access Control (MAC) protocols are responsible for managing radio communication that constitute the main energy consumer in wireless sensor-actuator networks. The Dual-Mode Adaptive MAC (DMAMAC) protocol is a recently proposed MAC protocol for process control applications in industrial automation. The goal of the DMAMAC protocol is to improve energy efficiency along with addressing real-time requirements for process control applications. The DMAMAC protocol switches between two operational modes that correspond to the two main states in process control: the transient state and the steady state. The state-switch is a safety critical function of the DMAMAC protocol (along with other functional properties) motivating the application of formal verification techniques. In this article, we use timed automata and the Uppaal tool to verify the design of the DMAMAC protocol. We have created a time-based model in Uppaal that constitutes a formal specification of the DMAMAC protocol. Using this model, we have successfully verified key properties of the DMAMAC protocol, thereby increasing confidence in the design of the protocol.

Model checking, Timed automata, Medium Access Control Protocols, Wireless Sensor Actuator Networks

1. INTRODUCTION

A Wireless Sensor Actuator Network (WSAN) (Akyildiz and Kasimoglu (2004)) consists of sensors and actuators that use radio to send, relay, and receive information. WSANs are used in a wide range of domains including process- and factory automation, smart home automation, and health-care. Feedback-based control loops that use wired or wireless solutions are collectively known as Networked Control Systems (NCS) (Hespanha et al. (2007)). NCS mainly use wired communication systems, but are increasingly adopting wireless communication. The salient feature of a wireless solution is the reduction in cost and size compared to the use of wired networks. The use of wireless communication, however, also has shortcomings and it has not yet become the de-facto replacement for wired solutions. The limitations of wireless solutions include low-bandwidth, energy efficiency, signal interference, and packet-loss. Energy efficiency in particular is an important concern when devices are battery powered.

Wireless solutions are made up of a collection of protocols that cater for different functions. Medium Access Control (MAC) is one of the functions that are critical to the proper operation of the entire WSAN. MAC protocols govern the communication and control the use of the radio on each node in the network. The radio module is the dominant consumer of energy in wireless nodes. The Dual-Mode Adaptive MAC (DMAMAC) protocol is a recently proposed MAC protocol in (Kumar S. et al. (2014)) for process control applications. The protocol is aimed to provide an energy efficient solution. The DMAMAC protocol was proposed for NCSs with real-time and energy efficiency requirements. In particular, it targets process control applications that fluctuate between two states of operation: steady and transient. Fig. 1 shows a typical process control with two states. The transient state corresponds to the process state with large and frequent change in measurements of physical quantities, resulting in a high data rate. The steady state refers to the process state with measurements contained within a controlled range of values, thus requiring less data transfer. An example is process control for chemical reactors. The varying physical quantities are temperature and pressure which are measured by sensors. This can be controlled by varying the inflow of chemicals to the chemical reactors and using coolants, controlled by actuators. The state-switch is a safety critical feature of the DMAMAC protocol and can benefit from formal verification to ensure proper functioning.

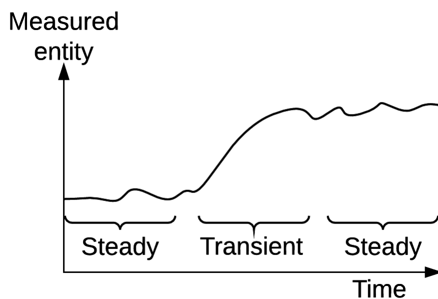


Figure 1: *Process control states*

Model-checking is a powerful technique for verification of protocol designs. Model-checking allows for exhaustive verification and has been widely used on related protocols (see, e.g., (Fehnker et al. (2012, 2007); Tschirner et al. (2008)). Verification in the early design phase can be used to ensure the behavioral correctness of protocols. Model-checking assists in discovering design faults by exhaustively traversing all possible execution traces of a given model. Furthermore, model-checking tools can provide error-traces to failure states, thus assisting in resolving any discovered design issues. Uppaal (David et al. (2011)) is a modelling and verification tool-suite that supports model checking of real-time systems. In addition to model-checking and verification, Uppaal also supports simulation which can be used to provide useful insights into the operation of a protocol.

In this article, we apply the Uppaal tool to analyze qualitative features of the DMAMAC protocol. We present a formal specification of the DMAMAC protocol in the form of a network of timed automata and verify safety properties related to the absence of faulty states. Additionally, we verify real-time properties including switch delay and maximum data delay. The timed modelling of the DMAMAC protocol is based on a Finite State Machine (FSM) representation of the sensors, actuators, and the sink node in the WSAN network configuration under consideration.

1.1. Related Work

Uppaal has been widely used to model and verify communication protocols (see, e.g., (Fehnker et al. (2012); Tschirner et al. (2008); Fehnker et al. (2007)). The Lightweight Medium Access Control (LMAC) (Fehnker et al. (2007)) protocol is the closest MAC protocol modelling related to the work presented in this article. The LMAC and the DMAMAC protocols are two distinct protocols with distinct goals, and differ significantly in their base features. The LMAC protocol is a self-organising protocol with nodes selecting their own slots i.e., time duration allocated for data transfer. The focus in the LMAC protocol verification is on efficient slot selection and collision detection. In the DMAMAC protocol, the slot scheduling is done statically and offline prior to deployment. The focus of the DMAMAC protocol is to provide an energy efficient solution along with efficient switching between the two operational modes. It requires a different model to represent the features of the DMAMAC protocol than the one used for the LMAC protocol. In (Tschirner et al. (2008)), the authors have focused mainly on modelling the Chipcon CC2420 transceiver. This work is related in terms of their use of a packet collision model and how collisions are observed. We use a collision model similar to (Tschirner et al. (2008); Fehnker et al. (2007)). With the extension of Statistical Model-Checking (SMC) features, Uppaal can also be used to assess performance related queries as shown in the case study (David et al. (2011)) of the Lightweight Medium Access Control (LMAC) protocol.

1.2. Outline

The rest of the article is organised as follows. In Sect. 2 we briefly introduce the DMAMAC protocol. For extensive details, we refer to (Kumar S. et al. (2014)). Section 3 describes in detail the constructed Uppaal model of the DMAMAC protocol. As part of this, we briefly introduce the constructs of timed automata as implemented in Uppaal, and perform some initial validation of the protocol model. In Sect. 4 we complete the validation of the constructed model. The verification of the protocol for different deployment configurations is discussed in Sect. 5. Finally in Sect. 6 we sum up the conclusions and discuss future work. The reader is assumed to be familiar with the basic concepts and operation of MAC protocols, including superframes and slots, and the principles of Time Division Multiple Access (TDMA) and Carrier Sense Multiple Access (CSMA).

2. DMAMAC PROTOCOL

The DMAMAC protocol (Kumar S. et al. (2014)) has two operational modes catering for the two states of process control applications: transient mode and steady mode. The protocol is based on Time-Division Multiple Access (TDMA) for data communication and a Carrier Sense Multiple Access (CSMA)-TDMA hybrid for alert message communication. The basic functioning of the protocol is based on the GinMAC protocol (Suriyachai et al. (2010)) proposed for industrial monitoring and control. The network topology of the DMAMAC protocol consists of sensor nodes, actuator nodes, and a sink. The sensor nodes are wireless nodes with sensing capability which sense a given area and update the sink by sending the sensed data. The actuator nodes are wireless nodes equipped with actuators, which act on the data performing a

physical operation. It is also possible to have wireless nodes with both sensors and actuators. The sink is a computationally powerful (relative to the nodes) wire powered node which collects the sensed data, performs data processing on it, and then sends the results to corresponding actuators.

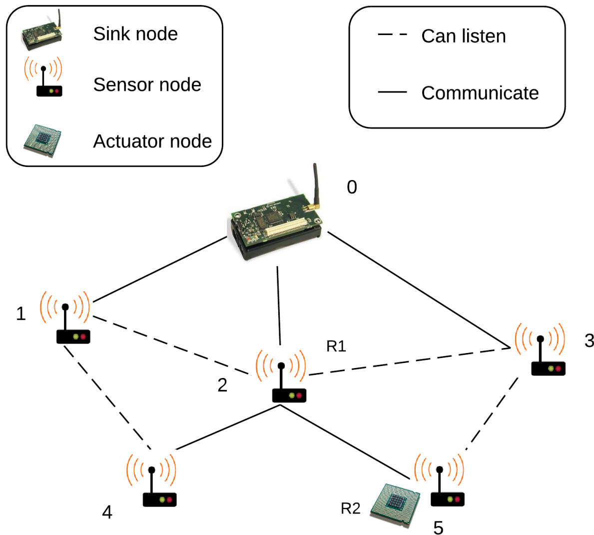


Figure 2: *The network topology for DMAMAC protocol*

Similar to the GinMAC protocol, the network deployment for the DMAMAC protocol is based on a tree topology as shown in Fig. 2. The solid lines between nodes represent data communication. The dashed lines represent nodes which can hear each other, but which have no direct data communication with each other. Each level in the tree topology is ranked (marked with “R#”, # is 1 or 2), with the sink having the lowest rank number and the farthest leaf nodes having the highest rank number. This ranking is exploited in the alert message sending procedure.

Firstly, we discuss the key assumptions that were made to support the design of the protocol. Further, we explain in brief the working of the two operational modes and the respective superframes they use.

- The nodes are assumed to be time synchronized via an existing time synchronization protocol. Thus, the time synchronization mechanism is not defined as a part of the protocol.
- The sink is assumed to be powerful, and it can reach all nodes in one hop.
- A pre-configured static network topology with no mobility is assumed.
- A single slot accommodates both a data packet and a corresponding acknowledgement.

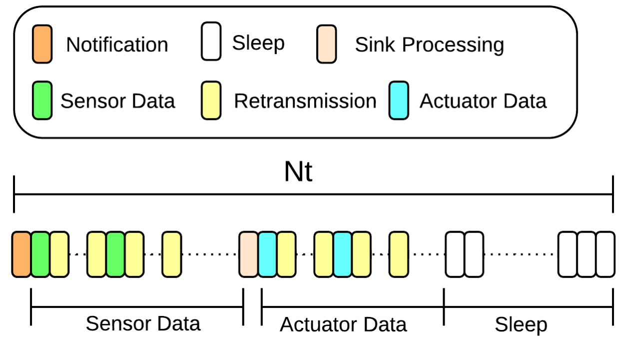


Figure 3: *The transient superframe of the DMAMAC protocol*

2.1. Transient mode

The transient mode is designed to imitate the transient state operation in process control. During transient state, the process changes rapidly generating data at a faster rate relative to the steady mode. During the transient mode operation, the DMAMAC protocol uses the transient superframe shown in Fig. 3. The superframe includes a data part for data transfer from the sensors to the sink, followed by a data part with data being sent from the sink to the actuators, and then a sleep part. The data part also includes a notification message slot from the sink to all nodes, and a sink processing slot. A typical transient mode operation cycle is described below:

- A notification message is sent from the sink to all the nodes. The notification message includes control data like state-switch message and time-synchronization. Time-synchronization is an integral part of TDMA based protocols.
- The data part is executed with data transmission from sensors to sink and then to actuators.
- The sleep part is executed where all sensors and actuators enter sleep mode in order to improve energy efficiency. This part represents the situation where all nodes are in sleep mode. Individually, the nodes are in sleep mode when they are not performing other tasks.

2.2. Steady Mode

The steady mode operation is designed to operate during the steady state of the controlled process. The steady superframe used in the steady mode operation is shown in Fig. 4. In addition to the parts that also exist in the transient superframe, the steady superframe contains an alert part. The alert part is used to ensure that the state-switch from steady to transient occurs whenever a sensor detects a threshold interval breach in its reading. This threshold is set by the sink when the switch from transient to steady is made. Note that w.r.t

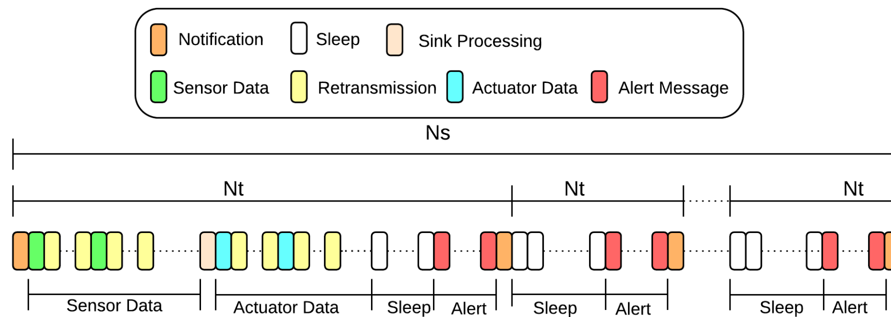


Figure 4: *The steady superframe of the DMAMAC protocol*

(Kumar S. et al. (2014)) a slightly modified steady mode superframe is used. There are notification slots placed at the end of each transient (Nt) part. This is done to facilitate immediate application of alert, and making a state-switch. In the alert part, one slot is allocated to each level or to nodes with the same rank. All the nodes in the same rank have the possibility to send an alert message in this slot. The alert sending method is described later. A typical steady mode operation cycle is as follows:

- A notification message is followed by the data and the sleep part, similar to the working in transient mode operation.
- (Alert part) Sensor nodes that have alert messages to be communicated use appropriate slots provided for each rank to notify parents about the alert. This is relayed towards the sink which then makes the switch to the transient state. In an absence of alert, sensor nodes still wake up on their alert receive slot and then enter sleep mode until the next notification slot.
- In the alert part, the notification slot is placed at the end. This is to ensure a quick transition between the two states. All regular nodes wake up in this slot, and receive a notification message from the sink. Alert notification to change superframes is sent here.

2.3. Change of superframes

A process switches between two states: transient and steady. The DMAMAC protocol follows these states via its transient and steady mode operation. There are two switches possible: transient to steady and steady to transient. The latter is a critical switch since the data rate in transient is higher and it is important to accommodate the higher data rate in transient state. The switch from transient to steady is decided by the sink, which determines if the process is in steady state based on previous readings. When the sink decides to make the switch, it informs all the nodes in the network to change their mode of operation. The message is sent via a notification message from the sink. When the sink node switches from transient to steady, it defines a threshold interval within which the sensor readings should lie, and informs the sensors about this threshold interval. During the entire steady mode operation, the sensors constantly monitor for a threshold breach. When there is a breach, the sensor node waits until its alert slot, then notifies its parent, which in turn forwards the alert towards the sink. The sink then informs the nodes in the network to switch to transient in its immediate next notification message.

2.4. Alert Message

An alert message is the message created by the sensor nodes to notify the sink that a state-switch is required. The sensor nodes choose a random delay in the slot before transmitting the alert message. At the completion of the time duration of the random delay, the nodes sense the channel to prevent collision. If a node during channel assessment detects another node sending an alert message, then it just drops its alert message. Collisions are still possible, e.g., when two nodes choose the same random delay or when two senders cannot listen to each other but the receiver can listen to both. Nodes check for a change of operational mode following the sending of the alert. If no change occurs (because of collision) the nodes save the alert and send the alert again in the next alert slot.

3. THE DMAMAC UPPAAL MODEL

Uppaal (David et al. (2011)) is a tool-set based on timed automata for model-checking of real-time systems. It is an integrated tool environment that supports modeling, simulation, validation, and verification. An abstract representation of a real-time system in the form of a model is structured as a network of timed automata. The query language of Uppaal allows for verification of safety, reachability, liveness, and time-bounded properties. In Uppaal, models are constructed as a network of templates based on timed automata. Templates are used to represent independent entities (e.g. a sensor node). Uppaal consists of two simulators: a symbolic and a concrete simulator. The symbolic simulator is used to

inspect the execution of the model step by step. For certain queries, Uppaal outputs traces which can be viewed in the symbolic simulator. This is useful for pin-pointing error locations and sequences of events that lead to errors/faults. The symbolic simulator also shows all the templates in the model and message sequence charts (MSC) can be used to visualize communication between different processes. The symbolic simulator also allows interactive step-wise simulation of the model. Along with features similar to the symbolic simulator, the concrete simulator has the added advantages of firing transitions at a specified time. For extensive detail on modeling in Uppaal, we refer to (Behrmann et al. (2004)).

3.1. Model design decisions and assumptions

We use a non-deterministic timed automata model to verify the properties of the DMAMAC protocol. The constructed model has several sources of non-determinism including the delay for sending alert messages in nodes, and the decision made by the sink to change from the transient mode to the steady mode. Given the design of alert messages, collisions are possible when sending alert messages. We use a simplified collision model, detailed later in this section. The sink and sensor/actuator nodes have separate timed automata models. Local clocks are used for each automaton. A global clock is used for a common network time reflecting the assumption on time synchronisation between the nodes. The main aim of the verification of the DMAMAC protocol model is to check that the two modes of operations are working correctly given the presence of non-deterministic choices (like collision) during execution and the delays that may occur.

Below, we discuss the assumptions and design decisions made during the construction of the Uppaal model for the DMAMAC protocol.

- Packets are abstractly modeled without pay-load. The messages or packets exchange mechanism is represented by channel syn-chronization in the Uppaal model.
- A time synchronization mechanism is provided using clock variables in Uppaal. This can be considered as a way of implementing the time synchronization between nodes assumed by the DMAMAC protocol.
- An exact model of CSMA results in a rather complex model. Instead, we use a representative CSMA procedure, which imitates the service and effects of actual CSMA on the working of the protocol. The effects include skipping packet transmission on detection of ongoing transmission and also collision. This makes our model and verification independent of the particular CSMA procedure that may be used in conjunction with DMAMAC.
- The collision caused due to the use of CSMA has effects on the state-switch procedure. A simple collision model is used, where we record collision when two or more nodes send packets at the same time. Collision results in failure of the packets, thus affecting the state-switch procedure.

A channel synchronization variable *choice* is used to force enabled transmissions. This is a modeling artifact and is not part of the protocol as such. In Uppaal, execution of models can stay in a location indefinitely even after outgoing edges are enabled. To force the model execution to continue via enabled outgoing edges, an urgent channel synchronization is required.

3.2. Sink Model

The sink model is shown in Fig. 5. We have used colors in the automaton locations to differentiate between states. Both the sink and the node automata begin in an initial location **Start**. The sink initiates the startup procedure of the network using a broadcast synchronization channel *startup* on the edge towards the **StateController** location. The function INITIALIZE() is used to set proper values to local and global variables. The sink reaches the **StateController** location upon having executed the startup procedure of the network. The node automata synchronize with the channel variable *startup*, and reach the **StateController** location.

The **StateController** location represents an event handler for handling transition between different states in the state-machine. The sink model uses a local clock variable "x", which is active in all states indicated by "$x' == 1$". "$x' == 0$" can be used to pause the clock counter. This is used as an invariant on all states to represent continuously running time. It also includes an invariant $x \leq currentMaxSlots * 10$ to prevent it from being in the state beyond the maximum timeframe of the active superframe (transient or steady). A typical slot time in WSANs is 10 milliseconds, and this we use the unit "ms" for time in our model. Given the time unit of "ms", the variable *currentMaxSlots* is multiplied by 10 to obtain the slot-time. The use of the **StateController** location is also similar to the *self-message* handler in the commonly used OMNeT++ (Varga and Hornig (2008)) framework for MAC protocols. The function of this particular handler is to check the self-message that it receives, and to act on the message by choosing an appropriate next state. Also, it determines the next state which is then sent as a self-message. The automaton changes between the different locations (states) in the model

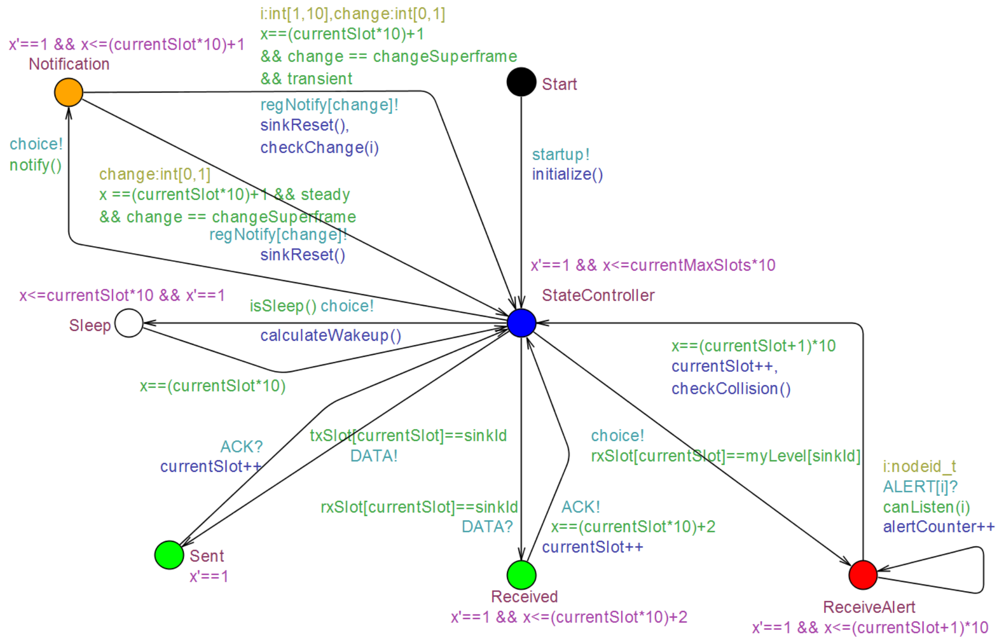


Figure 5: *Uppaal Model of the sink*

based on the local variable *currentSlot*, and the local clock variable x.

The **Notification** location is reached when the sink is due to send a notification message. The notification message is sent by the sink, and received by other nodes in the network. This is represented by the broadcast channel *regNotify[change]*, where *change* carries a message of the status of the Boolean variable *changeSuperframe*. The *changeSuperframe* variable is true when the sink needs to indicate to the nodes a change in superframes to switch the mode of operation. For both the steady to transient switch and the transient to steady switch, the sink uses *changeSuperframe*.

The switch from transient to steady is decided by the sink. There are two separate notification edges for transient mode and steady mode. In the transient mode, the sink decides if it has to switch to steady mode based on the random selection statement $(i : int[1, 10])$ and the obtained change value is sent over the channel. In the absence of real inputs a random selection is used. The edge with select statements $(i : int[1, 10])$ and $(change : int[0, 1])$ is used in transient mode. The second select statement $(change : int[0, 1])$ is a modeling artifact used to be able to send the value of *changeSuperframe* over the channel via the synchronization variable *regNotify[change]*. The guard $change == changeSuperframe$ makes sure that the select statement selects the same value as the *changeSuperframe* variable.

In the steady mode, a switch is based on alert from nodes. The notification for the steady mode is done via the edge with only one select statement $(change : int[0, 1])$. As a symbolic representation, we have used a guard $x == (currentSlot * 10) + 1$ on these edges, and an invariant $x \leq (currentSlot * 10) + 1$ on location **Notification** to indicate a delay of 1 ms for message transmission. Both these edges use a function SINKRESET(), which resets the sink variables at the beginning of a new superframe, and implements changing of superframes.

The **Sleep** location is reached when the sink or nodes do not have any active operations to be conducted in the current slot. The edge to **Sleep** is guarded by a Boolean function ISSLEEP() which checks if the current slot is a sleep slot. We use the urgent broadcast channel *choice* (model artifact) to force this transition whenever ISSLEEP() evaluates to true. In the absence of this channel variable, the model can continue to be in the location **StateController** forever even when ISSLEEP() is true. The location **Sleep** has an invariant $x <= currentSlot * 10$ which indicates that during execution, the control can be in the location as long as the time does not exceed the value *currentSlot*, which holds the value of the slot at which the sink should wake up for its next event. This is set by the function CALCULATEWAKEUP() when **Sleep** is reached.

The location **Sent** is reached when the sink sends data in its data slot. The **Received** location is reached when the sink receives any sensor data, and then sends an ACK via channel synchronization. Location **Received** has an invariant $x \leq (currentSlot * 10) + 2$. This invariant is used

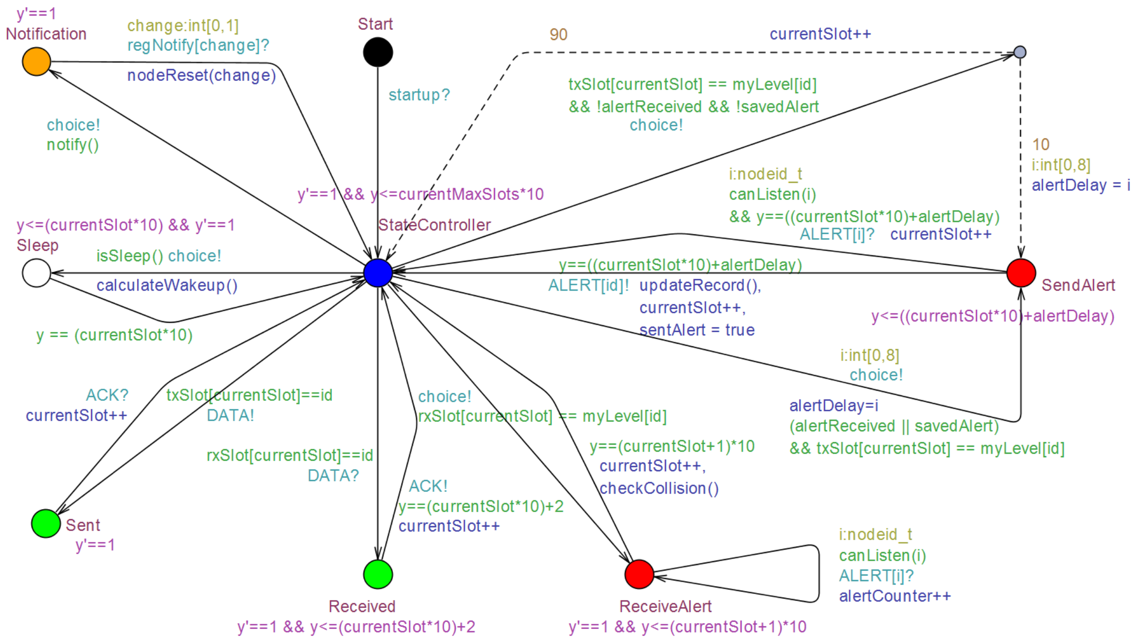


Figure 6: *Uppaal Model of a regular sensor/actuator node*

to add a delay of 2 ms as a representation for the time required for data communication. A follow up guard on the ACK sending edge $x == (currentSlot * 10) + 2$ makes sure that the delay is applied. Upon sending or receiving of ACK synchronization, the local variable *currentSlot* is incremented.

Lastly, the location **ReceiveAlert** is reached when it is the sink's turn to receive an alert. This is determined by the alert levels defined in the *myLevel* array variable represented by the guard $rxSlot[currentSlot] == myLevel[sinkId]$. The sink stays in the location for an entire slot duration (10 ms), and waits for any alerts from nodes it can listen to. The CANLISTEN(I) function is used as a guard to make sure that the sink listens to alerts from only those nodes that are in its listening range (same function used for nodes). The variable i given as input to the function is the result of a select statement $i : nodeid_t$, which allows the node to listen to any node that is transmitting. The guard makes sure that the sink can listen to that particular node. At the completion of the alert slot, the sink checks if any collision has occurred via the function CHECKCOLLISION(), and gives back the control to the **StateController**.

3.3. Sensor/Actuator Node Model

The sensor/actuator node model is shown in Fig. 6. The node model is similar to the sink model except for the notification handling procedure. Also, the node template consists of an extra location for sending alert messages. The notification part of the node model is simpler, since nodes only receive notifications. Location **Notification** is reached when the node is in its notification slot (receive) in either of the two types of superframes. The nodes then synchronize on the channel variable *regNotify[change]* from the sink, and reset the node variables using the function NODERESET() based on the value of the variable *change*.

The node model works similar to the sink model for sleep, sent (data), received (data), and alert receive. This means that in locations **Sleep**, **Sent**, **Received**, and **ReceiveAlert** the node and the sink model have the same modeling elements. Further, the location **SendAlert** which handles the crucial part of alert message sending is required by the protocol for the switch of operational mode from steady to transient. Based on the protocol specification, a node can send an alert message when the sensed data crosses the threshold interval. This threshold interval is set by the sink depending on the particular process being controlled. We imitate this event using a probability weight-based selection for sending alert messages as reflected in the edge towards **SendAlert** (shown with dashed lines). The edge with probability weight 90 represents no alert to be sent. The one with probability weight 10 represents the choice to send an alert. The guards on the edge make sure it is the alert slot of the node, and that the node does not already have an alert to be sent. When the node chooses to send an alert, a delay is chosen within the interval $[0, 8]$ via the select statement $i : int[0, 8]$. The chosen value is assigned to the *alertDelay* variable. The node then waits in the location **SendAlert** for the duration of the delay and performs carrier sense prior to sending the alert message. This is represented by the edge with the guard function

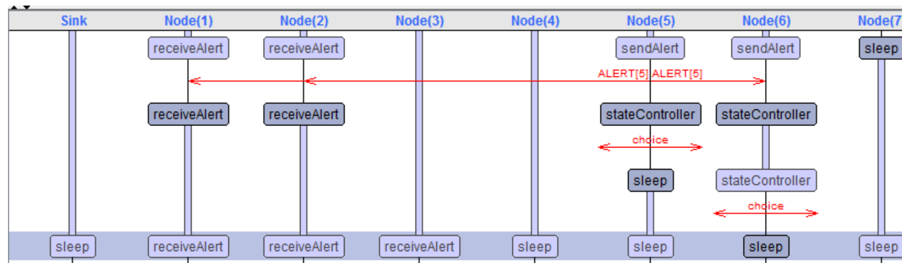


***Figure 7:** Message sequence chart for carrier sense during Alert*

Listing 1: Carrier Sense trace

```
(Sleep, receiveAlert , ReceiveAlert , ReceiveAlert , Sleep, StateController , StateController , Sleep)
choice: Node(5)[][3]−> // Delay of 3 ms chosen by node 5
(Sleep, ReceiveAlert , ReceiveAlert , ReceiveAlert , Sleep, SendAlert, StateController , Sleep)
choice: Node(6)[][3]−> // Delay of 3 ms chosen by node 6
(Sleep, ReceiveAlert , ReceiveAlert , ReceiveAlert , Sleep, SendAlert, SendAlert, Sleep)
ALERT[5]: Node(5)−> Node(1)[5]Node(2)[5]Node(6)[5] // Alert send by node 5 is heard by node 2 and node 6
(Sleep, ReceiveAlert , ReceiveAlert , ReceiveAlert , Sleep, StateController , StateController , Sleep)
```

CANLISTEN(I), where the node synchronizes to the broadcast channel *ALERT[i]* sent by other nodes in the vicinity (listening range) to skip sending a message.

We use a representative carrier sense mechanism in the model. Nodes skip sending an alert when another node within their listening range is sending with the same delay. In reality, carrier sense would involve listening to the channel for a small duration before sending the packets. Also, in a case where two nodes start carrier sense at the same instant, their packets would collide since they would start sending at the same instant after the carrier sense delay. In the carrier sense mechanism presented here, we represent a case in which two nodes can hear each other and have the same delay by one of the two nodes skipping the sending the alert message. When the nodes do not hear each other and the receiver can hear both, the packets collide at the receiver. An example message sequence of carrier sense is shown in Fig. 7. In this example, Node(5) and Node(6) are trying to send alert with the same delay (3ms) as shown in List. 1. In the listing, we have added comments with prefix "//" to add more detail. When Node(5) begins to send the alert, Node(6) senses the sending and skips sending alert via the edge guarded by CANLISTEN(I) function.

In a case where the channel is free, the nodes send an alert at the time instant after the chosen delay. The sending is represented using the send part of the broadcast channel variable *ALERT[id]!*. The local variable *currentSlot* is updated, along with the variable *sentAlert*, and function UPDATERECORD(). The variable *sentAlert* is used by the node to remember that it has sent an alert. In a case where no superframe change occurs after an alert was sent, a node updates its local variable *savedAlert*. The UPDATERECORD() function updates a global array variable *alertTimeRecord[]* which stores the delay chosen by each node in the given round. This is used to check if a collision has occurred. In certain cases when the alert messages fail to reach the sink due to collision, the *savedAlert* variable is used to save the alert, that is sent again in the next round. During this, the probability edge is not used. Instead, the nodes directly move to the location **SendAlert** via the edge (solid line) with the guard $(alertReceived||savedAlert)$. The variable *alertReceived* represents the case when nodes have to forward an alert received from other nodes towards the sink. The nodes then choose a new delay value from the interval $[0, 8]$ for sending the alert message again.

3.4. Collision

We use a simple collision model similar to the one used in (Tschirner et al. (2008)) and (Fehnker et al. (2007)). In the LMAC (Fehnker et al. (2007)) protocol, when two nodes send a packet in the same slot it is considered as a collision. In the DMAMAC protocol model, collision is counted when a node receives at least two alert messages with the same delay in the same alert slot. In our model, we assume that apart from its child nodes, the parents can also listen to nodes in the vicinity (similar to real networks). We define statically which other nodes a given node can listen to. Based on the representative carrier sense model, the collision occurs at a node only when it receives two alert messages from nodes (of the same rank) that cannot listen to each other, and had chosen the same delay within the alert slot. A message sequence chart showing a collision occurrence at the sink is shown in Fig. 8.

The trace corresponding to the Message Sequence Chart in Fig. 8 is shown in List. 2. In the considered scenario, Node(1) and Node(3) choose the same delay of 7 (ms) independently. Since they cannot listen to each other their alert packets end up

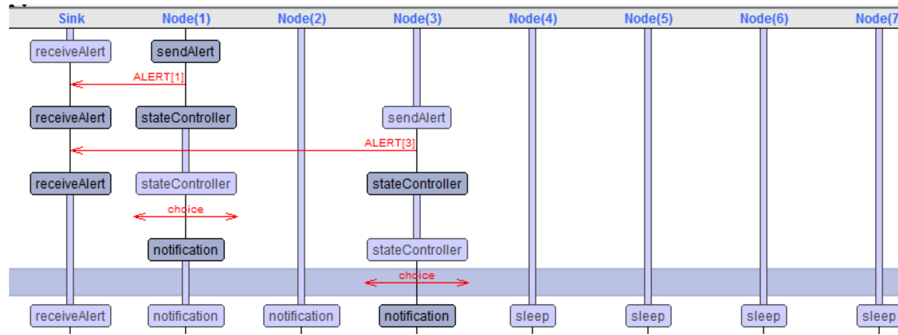


Figure 8: *Message sequence chart for collision at the sink*

Listing 2: Collision Trace

```
(ReceiveAlert, StateController, Notification, StateController,Sleep,Sleep,Sleep,Sleep)
choice: Node(1)[][7]-> // Delay of 7 ms chosen by node 1
(ReceiveAlert,SendAlert, Notification, StateController,Sleep,Sleep,Sleep,Sleep)
choice: Node(3)[][7]-> // Delay of 7 ms chosen by node 3
(ReceiveAlert,SendAlert, Notification,SendAlert,Sleep,Sleep,Sleep,Sleep)
ALERT[1]: Node(1)-> Sink[1] // Alert sent by node 1 to the sink
(ReceiveAlert, StateController, Notification,SendAlert,Sleep,Sleep,Sleep,Sleep)
ALERT[3]: Node(3)-> Sink[3] // Alert sent by node 3 to the sink
(ReceiveAlert, StateController, Notification, StateController,Sleep,Sleep,Sleep,Sleep)
```

colliding at the sink. This prevents a change of the superframe (mode of operation). Node(1) and Node(3) detect this and save the alert. The saved alert is used to resend the alert in the next round (with a new delay) to make sure the superframe changes. Note that there could be a situation where collision occurs at lower levels (and even at the sink), but still the change of superframe occurs because of another alert message reaching the sink. For our model, we have created the topology in such a way that both cases exist in different configurations as discussed in Sect. 5. In reality, the receiver nodes do not detect collision: in certain cases nodes receive parts of packets that collide (difficult to decode them) and in other cases they receive nothing at all. In that respect, we rely on a representative model of collision detection designed to be consistent with the effects of collision on the change of superframe.

3.5. Network Topology

The node topology used for the verification of the models is shown in Fig. 9. We use 5 sensor nodes, 2 sensor-actuator nodes, and a sink in the tree topology considered. We consider a small topology to keep the state-space small which is needed in order to conduct exhaustive verification. The current node topology has 3 ranks but since the sink only listens (and does not send alerts), we have 2 alert slots in the DMAMAC protocol. This representative topology allows for both carrier sense and collision, has both sensors and actuators with data communication for both types of nodes, and also has multiple hops. A topology based assumption for listening range is that a higher level (lower rank) node is listening to all of its children nodes, and also sometimes to other nodes in the vicinity. A real node has a listening range based on its receiver sensitivity, and distance with other nodes in the vicinity that varies with topology. Given that our main aim is to check the working of the protocol, we define the listening range in the topology manually instead of calculating it dynamically based on multiple factors like node position, path-loss, and receiver sensitivity as is typically done in network simulators for quantitative analysis.

In the topology used for the evaluation of the DMAMAC protocol the functionalities that need to be verified are covered. The DMAMAC protocol is used for applications with offline scheduling. This means that scheduling is done prior to deployment and all slot allocations are known prior to deployment. The topology in general is well-planned, and no random deployment is used. A real topology would be much larger than the one considered here. In the current topology, we have 3 levels and a maximum of 2 hops. For a qualitative analysis this covers the error scenarios that could potentially exist with multiple-hops.

The schedule for the considered node topology is shown in Fig. 10. The schedule shows both sender/receiver identification (node/sink). The sending part is marked by TX and receiving part is marked by RX. For notification messages, only the sender identification (sink) is marked. The schedule only represents the steady superframe. In the transient superframe only the first *Nt* part is used with the alert parts replaced by sleep. Note that we use 10 milliseconds (“ms”) as slot duration similar to slot sizes used in general practise.

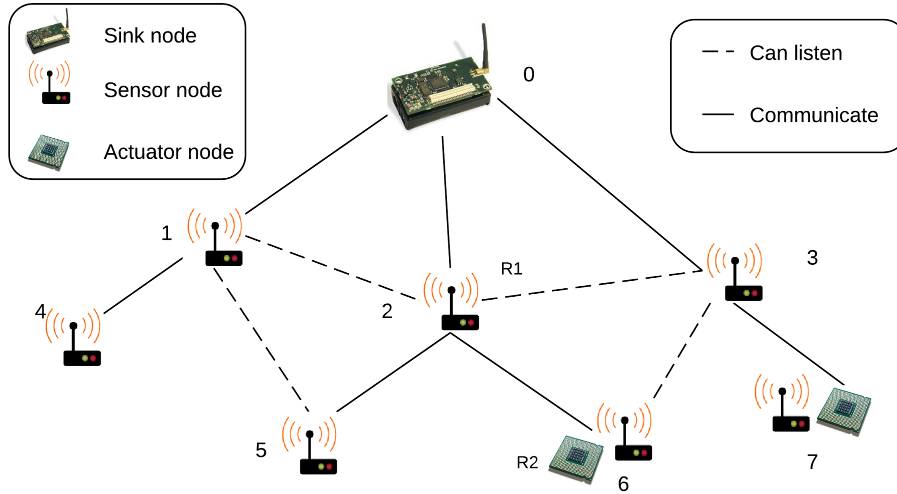


Figure 9: *Node topology*

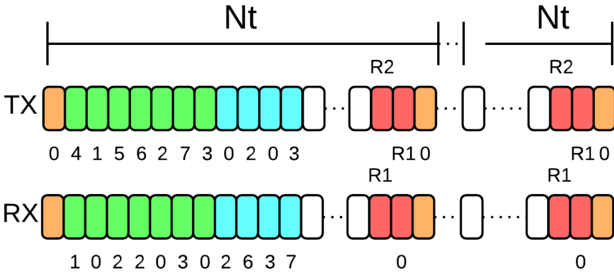


Figure 10: *Superframe structure based on the schedule and node topology*

3.6. Configurations

Multiple configurations of the DMAMAC protocol can be analysed based on values that can be varied in the model. Firstly, the Uppaal model can start in either steady or transient mode and this could have an effect on some of the verification properties (as discussed in Sect. 5). Another important factor affecting the configurations is the range of possibilities for the variable *alertDelay*. In the protocol, we have used the range [0,8] to reduce collision. Due to state-space issues we use only $alertDelay[1,1]$ for exhaustive queries, e.g., deadlock query. The $alertDelay[1,1]$ in itself covers all possibilities including possibility of state-switch, collision and CSMA, and hence all the qualitative aspects of the protocol. For other non-exhaustive queries, we use up to $alertDelay[1,4]$ configurations to further validate the verification procedure. The only difference between $alertDelay[1,1]$ and the other considered configurations is the applicability of property *sink mode* and *consistent node mode* of the verification properties and is further discussed in Sect. 5. Also, the select statement interval $[1,10]$ used to decide the switch from transient to steady mode by the sink is reduced to $[1,2]$ to keep the state-space low for all the queries. The reduction of the interval only means that in transient mode there is a 50% probability to switch to steady mode, and thus does not affect the qualitative results.

4. MODEL VALIDATION

We first validate the constructed Uppaal model of the DMAMAC protocol by checking some basic behavioural properties related to the operation of the model. The purpose is to obtain a high degree of confidence in the constructed model prior to verifying key properties of the protocol in the next section. During construction of the DMAMAC protocol, we validated the operation of the model via MSCs obtained from step-by-step execution of the model in the Uppaal simulator. These properties were related to data transmission between nodes, data transmission between the nodes and the sink, sending/receiving of alert message, possibility of collision, and carrier sense when sending alert messages.

Below we validate properties of the model related to data communication and collisions using the verification engine of Uppaal. For this, we express the properties to be validated in the form of Uppaal queries. Queries in Uppaal are written in a restricted variant of Computation Tree Logic (CTL) in which path formulas cannot be nested. Specifically, the following path formulae are supported by Uppaal: A$\Box$ (always globally), A$\Diamond$ (always eventually), E$\Diamond$ (reachable), and E$\Box$ (exists globally).

For validation purposes, we first check the operation of the model with respect to data communication between neighbouring nodes and between the sink and its neighbouring nodes. We check that if two nodes i and j are such that the parent node of node j is i, then these will eventually communicate. Furthermore, it should be such that

any child node of the sink node should eventually communicate with the sink. Formally, these two properties can be expressed as the set of queries below. Here, $nodeid_t$ is the type used to represent node identifiers in the model, $parent[i]$ is used to obtain the parent node of node i, and $sinkId$ denotes the identity of the sink. The property is expressed by reference to the location **Sent** and location **Received** which are reached by the communicating nodes upon synchronization over the channel *DATA*.

Node data communication
$\forall$ i,j $\in$ nodeid_t such that parent[j]==i:
A$\Diamond$ (Node(i).Sent && Node(j).Received)

Sink data communication
$\forall$ i $\in$ nodeid_t such that parent[i]==sinkId:
A$\Diamond$ (Node(i).Sent && Sink.Received)

It should be noted that we do not check the property that two neighbouring nodes always have the possibility to communicate. This is due to the fact that Uppaal does not support nesting of CTL path formulae.

The second property that we validate is related to collisions which play an important role in the DMAMAC protocol in relation to the sending of alert messages. In this case, we check that it is possible to have collision happening on all nodes and on the sink. Collision cannot be guaranteed to happen and hence we verify only the possibility of collision occurring. Formally, these two properties are expressed as the following set of queries:

Node collisions $\forall$ i $\in$ nodeid_t : E$\Diamond$ Node(i).collision

Sink collisions E$\Diamond$ Sink.collision

Finally, we also validate that there are no deadlocks in the model. In Uppaal, this can be expressed via the query below where *deadlock* is a built-in state property in Uppaal.

No deadlock A$\Box$!deadlock

The above queries related to data communication, collision, and deadlocks were all verified on both the transient and the steady variant of the model. This in turn increased confidence in the proper operation of the model.

5. PROTOCOL VERIFICATION

We now consider verification of the key functional properties of the DMAMAC protocol. As explained earlier, the constructed model comes in two variants: one variant with the protocol starting in the transient mode and one variant with the protocol starting in the steady mode. We first consider common properties that are independent of whether the protocol starts in the transient or in the steady mode. Then we consider properties specific for the transient mode case followed by properties specific for the steady mode case. Finally, we verify two real-time properties of the protocol related to upper bounds on mode switch delay and data transmission delay.

5.1. Common Properties

Given the dual-mode operation of the DMAMAC protocol, the important properties relate to the nodes operating in different modes, and switching between them. Firstly, we check the operating mode properties. We make sure that the sink is exclusively either in the steady mode or in the transient mode at all times. Following this, we check that all nodes follow the operating mode of the sink consistently. Formally, these properties are expressed as follows:

Sink mode
A$\Box$ (Sink.steady && !Sink.transient) $||$ (!Sink.Steady && Sink.transient)

Consistent node mode
$\forall$ i $\in$ nodeid_t:
A$\Box$ (Node(i).transient $==$ Sink.transient $||$ Node(i).steady $==$ Sink.steady)

Next, we investigate properties of the protocol related to collision and its effect on the change of operational modes. The queries refer to the *changeSuperframe* variable which indicates whether the network should change mode in the next superframe. Collisions may have different effects depending on the configuration under consideration. For configurations with $alertDelay[1,1]$ where all the nodes will pick the same delay, collision at the sink should not result in a change of superframe or operational modes. For configuration with $alertDelay[1,2]$, we may have both collision and change of superframe since, e.g., two nodes may pick a delay of 1 (which will result in a collision) while a single third node picks a delay of 2. The latter choice will result in the sink being notified of a required change of mode. Formally, properties related to collisions and change of mode are specified as follows:

Collision and mode switch
E$\Diamond$(Sink.collision && Sink.changeSuperframe)

Collision and no mode switch
E$\Diamond$(Sink.collision && !Sink.changeSuperframe)

Following the discussion above, we expect that the first property is false in configurations where all nodes must choose the same delay while it is true in configurations where different alert delays can be chosen. This implies that the protocol design ensures that the DMAMAC protocol can change mode even in the presence of collisions. The second property is expected to be true as we may (in all configurations) have the situation that the choice

Listing 3: Collision and change of superframe together

```
(ReceiveAlert, StateController, StateController, StateController, Sleep, Sleep, Sleep, Sleep)
choice: Node(1)[][1]-> // Node 1 chose delay of 1 ms
(ReceiveAlert, SendAlert, StateController, StateController, Sleep, Sleep, Sleep, Sleep)
choice: Node(3)[][1]-> // Node 3 chose delay of 1 ms
(ReceiveAlert, SendAlert, StateController, SendAlert, Sleep, Sleep, Sleep, Sleep)
choice: Node(2)[][2]-> // Node 2 chose delay of 2 ms
(ReceiveAlert, SendAlert, SendAlert, SendAlert, Sleep, Sleep, Sleep, Sleep)
ALERT[3]: Node(3)->Sink[3] // Node 3 sends alert
(ReceiveAlert, SendAlert, SendAlert, StateController, Sleep, Sleep, Sleep, Sleep)
ALERT[1]: Node(1)->Sink[1] // Node 1 sends alert
(ReceiveAlert, StateController, SendAlert, StateController, Sleep, Sleep, Sleep, Sleep)
ALERT[2]: Node(2)->Sink[2] // Lastly, Node 2 sends alert with a delay of 2 ms (higher than others two)
(ReceiveAlert, StateController, StateController, StateController, Sleep, Sleep, Sleep, Sleep)
```

of delay (alert) causes collisions such that the sink may not be notified of the required change of mode in the current superframe. Of course, the sink may be notified via retransmission of the alert in a later superframe, eventually causing a mode switch (see below).

An example trace demonstrating co-existence of collisions and change of superframe is shown in List. 3. In this example, the nodes 1 and 3 choose the same delay (1 ms) and cannot listen to each other. The transmissions therefore collide at the sink. But node 2 which has chosen a different delay (2 ms) successfully alerts the sink thus inducing change of superframe. This delay choice is done when the model changes location from **StateController** to **SendAlert**.

Finally, we verify a property related to the critical change of state in the protocol from steady to transient. When the data requirement is higher, the protocol should be able to detect and switch accordingly. Also, when a switch fails due to collisions, there should be a possibility to re-use the failed alert to induce change of operational modes. The failed alert is used as a saved alert in the next alert round. We use the query which searches for one example where this occurs.

Critical change of state
$\exists$ i $\in$ nodeid_t:
E$\Diamond$ Node(i).savedAlert && Sink.steady && Sink.changeSuperframe

It should be noted that given the nature of the model, the collisions could occur forever preventing the change of superframe. This means that we cannot show that a state switch will eventually happen.

5.2. Transient model variant

We now consider properties specific for the variant of the model where the sink and nodes starts in transient mode. In the model, transient and steady are boolean variables. In the case where the controller process stays in the transient state permanently, the protocol needs to stay in transient mode of operation to suit the application needs. The given query below checks if there is a path where the system invariantly is in the transient mode.

Remain transient E$\Box$ transient

The second property represents the reachability of steady mode from the starting state, i.e., that it is possible for the system to change mode from transient to steady.

Steady switch E$\Diamond$ steady

5.3. Steady model variant

For the variant of the model that starts in the steady mode, we verify the dual properties of the variant that starts in the transient mode. These two properties are listed below:

Remain steady E$\Box$ steady

Transient switch E$\Diamond$ transient

The two properties check that it is possible to remain in steady mode and that it is possible to switch to transient mode.

5.4. Real-time properties

We now consider real-time properties related to mode switch delay and data communication delay. In order to verify these properties, we use a modified version of the Uppaal model where we have included the use of two watch templates (Watch1 and Watch2) in order to record elapsed time.

The first real-time property that we consider is the switch delay, i.e., the time difference between a detection of threshold breach and the mode switch happening. This switch is required to happen within the duration of a superframe (transient superframe length). This property is specified as follow:

Switch delay
A$\Box$ Watch1.switchDelay $\leq$ superframeLength

We verified the switch delay property by considering a single node farthest from the sink. By symmetry, the property applies to other nodes at the same level, and also to the parent nodes which (by the tree topology) will have a smaller maximum switch delay.

The second real-time property concerns the data communication delay. It is the time elapsed between the first data sent in the superframe until the last data received. This is required to be within the same superframe. The property is expressed as follows:

Data delay

A□ Watch2.dataDelay $\leq$ superframeLength

All properties listed above evaluate to the expected results. Details on the execution time for the queries based on different configurations of the model are shown in table 1. The verification was conducted on a PC with 4 GB RAM, 2.30 GHz 2-core processor. The query **Collision and mode switch** could be verified only on the configuration with $alertDelay[1, 1]$ and resulted in memory exhaust in other configurations. Other queries were verified also on configuration $alertDelay[1, 2]$, and $alertDelay[1, 4]$ with $i : int[1, 2]$.

6. CONCLUSION AND PERSPECTIVES

In this article, we have detailed the modelling and verification process of the DMAMAC protocol. The DMAMAC protocol is designed for process control applications and we have used the Uppaal model-checking tool for modelling and verification. The model consists of a network of timed automata with multiple nodes and a sink operating according to the DMAMAC protocol.

We have explained the model in Uppaal including its modelling elements and templates in detail. The constructed timed automata model includes generic MAC slot operations including data sending and receiving, notification, and sleep. This means that the model can be extended to represent other MAC protocols with similar (and extra) slotting within their superframe. To illustrate the generality of the constructed model, the finite state machine for the protocol model and the possible extensions are shown in Fig. 11. The diagram is divided into three parts: the generic part, DMAMAC extensions, and other possible extensions. The generic part consists of notification and data transfer, which is generally part of a wide range of MAC protocols for WSANs. The DMAMAC extensions with alert sending and receiving parts are specifically relevant for DMAMAC. Given the generic structure, the model can be extended to include other MAC protocol slot types or state types including *Channel Sense*, *Backoff* and *Link establishment*. S-MAC (Ye et al. (2002)) is a one such MAC protocol that uses Request To Send (RTS), Clear To Send (CTS), and carrier sense. The current model can be easily extended to model S-MAC with re-use of generic parts.

We have validated the basic operation of the constructed model using message sequence charts highlighting the most important features, and operations of the protocol including data transfer, alert message functioning, carrier sense, and possibility of collision. Further, we validated the proper operation of the model using the verification engine of Uppaal. The validated model was then verified for the switch procedure and safety properties, including absence of deadlock and other faulty states. The key real-time properties in the form of upper bounds on switch delay and data delay were also verified. Two variants of the model were used for verification and validation, one starting with the transient mode of operation and the other starting with steady mode. Different configurations of the model with varying alert delay were used as a basis for the verification. For the verification, we used a representative node topology that covers all important features of the protocol including existence of sensors and actuators, multi-hop, alert messages, and possibility of collision. The DMAMAC protocol model in Uppaal satisfied the properties considered which increases confidence on the design of the protocol. As a proposed future work, a stochastic model of the DMAMAC protocol to verify the quantitative properties including collision probability, expected switch delay, and energy consumption could provide further insights to the working of the protocol.

Currently, we are also in the process of developing a prototype implementation of the DMAMAC protocol on real hardware. The Uppaal model constructed in this paper serve as an important specification in terms of ensuring the proper and correct implementation of the protocol logic and frame processing.

REFERENCES

Akyildiz, I. F. and Kasimoglu, I. H. (2004), ‘Wireless Sensor and Actor Networks: Research challenges’, *Ad Hoc Networks* **2**(4), 351–367.

Behrmann, G., David, A. and Larsen, K. (2004), A tutorial on Uppaal, *in* M. Bernardo and F. Corradini, eds, ‘Formal Methods for the Design of Real-Time Systems’, Vol. 3185 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 200–236.

David, A., Larsen, K. G., Legay, A., Mikucionis, M., Poulsen, D. B., Vliet, J. and Wang, Z. (2011), Statistical Model Checking for networks of Priced Timed Automata, *in* ‘Proceedings of the 9th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)’, Vol. 6919 of *LNCS*, Springer Berlin Heidelberg, pp. 80–96.

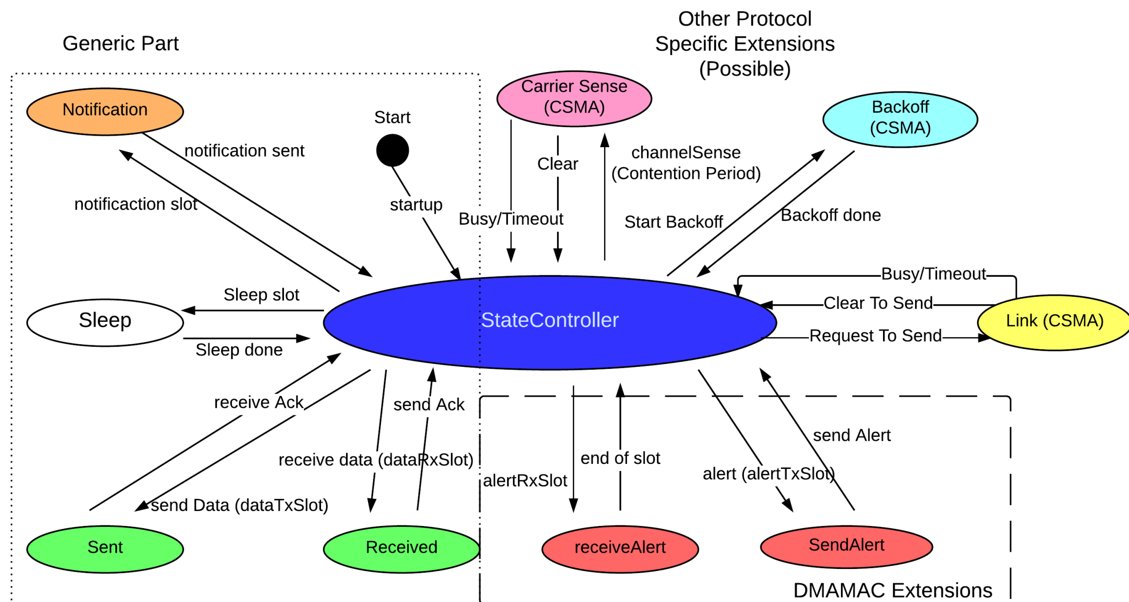


Figure 11: *Generic model with possible extensions*

Fehnker, A., van Glabbeek, R., Hfner, P., McIver, A., Portmann, M. and Tan, W. (2012), Automated analysis of AODV using Uppaal, *in* 'Tools and Algorithms for the Construction and Analysis of Systems', Vol. 7214 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 173–187.

Fehnker, A., van Hoesel, L. and Mader, A. (2007), Modelling and Verification of the LMAC protocol for Wireless Ssensor Networks, *in* J. Davies and J. Gibbons, eds, 'Integrated Formal Methods', Vol. 4591 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 253–272.

Hespanha, J. P., Naghshtabrizi, P. and Xu, Y. (2007), A survey of recent results in Networked Control Systems, Vol. 95, pp. 138–162.

Kumar S., A. A., Øvsthus, K. and M. Kristensen, L. (2014), Towards a Dual-Mode Adaptive Mac Protocol (DMA-MAC) for feedback-based Networked Control Systems, *in* 'The 2nd International Workshop on Communications and Sensor Networks'.

Suriyachai, P., Brown, J. and Roedig, U. (2010), Time-critical data delivery in Wireless Sensor Networks, *in* 'Proceedings of DCOSS', pp. 216–229.

Tschirner, S., Xuedong, L. and Yi, W. (2008), Model-based Validation of QoS properties of Biomedical Sensor Networks, *in* 'Proceedings of the 8th ACM International Conference on Embedded Software', EMSOFT '08, ACM, New York, NY, USA, pp. 69–78.

Varga, A. and Hornig, R. (2008), An overview of the OMNeT++ simulation environment, *in* 'SIMUTOOLS', pp. 60:1–60:10.

Ye, W., Heidemann, J. and Estrin, D. (2002), An energy-efficient mac protocol for wireless sensor networks, *in* 'INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE', Vol. 3, pp. 1567–1576 vol.3.

Property / Query	Result	CPU Time (s)	Resident Mem. (KB)	Virtual Mem. (KB)
Configuration : alertDelay[1,1], i:[1,2]				
Common queries				
Sink mode	Not Satisfied	718.837	1,795,596	3,595,148
Consistent node mode	Satisfied	876.586	1,772,436	3,573,820
Collision and mode switch	Not Satisfied	711.287	1,797,488	3,599,136
Collision and no mode switch	Satisfied	3.214	21,044	49,512
Critical change of state	Satisfied	33.868	113,084	238,116
Transient specific queries				
Remain transient	Satisfied	0.015	13,820	56,924
Steady switch	Satisfied	0.64	13,888	40,168
Steady specific queries				
Remain steady	Satisfied	0.032	17,424	58,892
Transient switch	Satisfied	1.965	19,308	48,796
Real-time queries				
Switch delay	Satisfied	273.048	440,676	891,152
Data delay	Satisfied	231.302	450,664	905,284
Configuration: alertDelay[1,2], i:[1,2]				
Common queries				
Sink mode	N/A	N/A	Memory exhausted	Memory exhausted
Consistent node mode	N/A	N/A	Memory exhausted	Memory exhausted
Collision and mode switch	Satisfied	8.331	62,292	128,008
Collision and no mode switch	Satisfied	8.346	61,616	129,064
Critical change of state	N/A	N/A	Memory exhausted	Memory exhausted
Transient specific queries				
Remain transient	Satisfied	0.02	13,836	40,092
Steady switch	Satisfied	0.562	13,848	57,012
Steady specific queries				
Remain steady	Satisfied	0.046	36,596	82,700
Transient switch	Satisfied	16.708	66,116	137,096
Real-time queries				
Switch delay	Satisfied	1200.225	1,799,596	3,601,164
Data delay	Satisfied	1068.014	1,799,624	3,622,604

***Table 1:** Performance of the protocol verification using Uppaal*

On quantitative Analysis of Time Open Workflow Nets and Parametric Extension

Zohra Sbaï
Université de Tunis El Manar
Ecole Nationale d'Ingénieurs de Tunis
BP. 37 Le Belvédère
1002 Tunis, Tunisia
zohra.sbai@enit.rnu.tn

Kamel Barkaoui
Laboratoire CEDRIC
Conservatoire National des Arts et Métiers
292 rue Saint Martin
Paris Cedex 03, France
kamel.barkaoui@cnam.fr

Collaborative systems design is today a complex process especially where constraints such as tasks delays or resources handling have to be considered. In addition to a well constructed model of a workflow system, a prior and rigorous verification phase is important to ensure a correct execution of the system. In this direction, we are interested in this paper by the modeling and the analysis of interacting processes in particular those constrained by timing delays. First, we present the Time open Workflow nets (ToWF-nets) which stand for a sub-class of Petri nets dedicated to model any business process which interact with other partners via interface places in order to meet a common goal. Then, after recalling the semantics of ToWF-nets, we investigate in presenting our recent results in quantitative verification of properties related to the correct communication of various ToWF-nets. We show how to make a TCTL based model checking of the studied properties. Finally, we extend our analysis approach of ToWF-nets to cover concurrent processes leading thus to propose a parametric verification of ToWF-nets.

Time Open Workflow Nets, Collaborative Systems, Quantitative Analysis, Timed Computational Tree Logic, Parametric Verification

1. INTRODUCTION

Nowadays, collaboration in organizations is more and more sought since it allows to end users to benefit from more enhanced and complex services. Although it is possible to allow complex services with a single organization, it is almost impossible to provide, in this case, simple services nor to reuse them. So, it is important to provide simple services and to allow their composition, if necessary, for more complex tasks. For instance, manufacturing systems can be seen as a network of servers and queues. This system process can be seen as a composition of multiple services interacting together to fulfill a unique goal. As an example, we mention a factory which make shirts. The first stage in shirt manufacturing is the cutting of the material to different shapes. Then, the next stage is to sew the pieces together. Finally buttons and a quick iron are added. This composition is more and more studied when treating inter organizational processes. We mention for example a manufacturing enterprize which has to communicate with clients, suppliers, etc. in order to fulfill the manufacturing task. Thus the notion of services composition is of great need and that's why it is very studied in academic as well as industrial environment.

In collaborative systems in general, different partners, having their own processes, interact together in order to achieve a common goal. We focus in this paper on the modeling and the analysis of such systems involving multiple communicating processes. In the modeling phase, we propose to use Time Open Workflow Nets (ToWF-nets), a sub class of Time Petri Nets (TPN) to model workflow processes with timing delays and with ability to communicate with partners. A ToWF-net consists of an open workflow net (oWF-net) in which we associate with each transition a minimum and a maximum amount of time needed to its execution. An oWF-net is a workflow net augmented with special places called interface places and used to interact with other processes.

For the purpose of analysis, we propose to enhance a formal verification due to solid theoretical basis of formal methods. More precisely, we adopt an analysis method based on the well known model checking techniques. In fact, Model checking is an automated verification technique for proving that a model satisfies a set of properties specified in

temporal logic. Given a concurrent system Σ and a temporal logic formula φ, the model checking problem is to decide whether Σ satisfies φ. Hence, we have to formulate in temporal logic the properties to be verified. This kind of verification is situated at the design phase, allowing thus to find bugs as early as possible and therefore to reduce the cost of failures. This, especially, permits us to check as early as possible if two or more processes are compatible before their composition. We express the proposed properties in Timed Computation Tree Logic (TCTL).

The rest of the paper is organized as follows. Section 2 is dedicated to the presentation of ToWF-nets for modeling interacting processes which are constrained by timing delays. In this section, we present the semantics of the model in terms of states and the states evolution. In section 3, we highlight quantitative analysis results of ToWF-nets. We recall first the principle of TPN-TCTL temporal logic, then we characterize some properties of interest and express them in this temporal logic and finally, we detail how to model check these properties via several examples. We focus in section 4 on a parametric verification of ToWF-nets composition and on some experiments in Romeo model checker. Finally, section 5 presents related work and section 6 concludes the paper with a discussion and a proposition of future work.

2. TIME OPEN WORKFLOW NETS

This section is dedicated to present Time open Workflow nets (ToWF-nets) : a Petri nets sub class allowing to model workflow processes communicating with partners via interface places. After defining ToWF-nets model and semantics, we expose some results on reachability analysis.

2.1. ToWF-nets to model communicating processes

To model a composition of interacting processes, we refer to Petri nets due to their expressive power as well as their theoretical basis. The well known Petri net class used in this modeling is named open workflow nets (oWF-nets) (Karsten and Schmidt (2005); Sbaï and Barkaoui (2013); Martens (2005); Massuthe et al. (2005)). Each involved process is modeled by an oWF-net possessing interface places which serve for the communication with other processes. In this way, the interaction and conversation between the different processes are guaranteed. The interaction considered here is ensured through operational and control behaviors. The operational behavior is specific to each process according to its business logic and the control behavior is ensured by the general behavior of any partner in the composite process.

As mentioned above, oWF-nets nets are mainly the extension of workflow nets (WF-nets) to model workflow processes which interact with other processes via interface places. Simple WF-nets (Sbaï and Barkaoui (2013, 2012)) is a result of Petri nets' application to workflow management. The choice of Petri nets is explained by the fact that they form a powerful formalism expressing the control flow in business processes (van der Aalst (1996); Esparza et al. (1989); Ellis et al. (1995); De Michelis et al. (1994)).

Commonly, a Petri net is a 4-tuple $N = (P, T, F, W)$ where *P* and *T* are two finite non-empty sets of places and transitions respectively, $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, and $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the weight function of *N* satisfying $W(x, y) = 0 \Leftrightarrow (x, y) \notin F$. If $W(u) = 1 \; \forall u \in F$ then *N* is called an ordinary net and it is denoted by $N = (P, T, F)$. For every node $x \in P \cup T$, the set of input nodes of x is defined by $^\bullet x = \{y | (y, x) \in F\}$ and the output nodes of x form a set denoted by $x^\bullet = \{y | (x, y) \in F\}$. We refer the reader to (Barkaoui et al. (2007)), for more Petri nets notations used in this paper.

A Petri net which models a workflow process is called a WF-net (van der Aalst (1997)). An ordinary Petri net $N = (P, T, F)$ is a WF-net iff N has one source place i named initial place (containing initially one token) and a sink place f named final place. In addition to this characteristic, every node in a WF-net $n \in P \cup T$ exists in a path from i to f.

For processes composition, we propose to model each process by a WF-net describing the tasks to be performed as well as their routing. Then, the conversation between the involved processes is ensured by the communication places used for the conversation and/or the messages sending. Thus, we are using open WF-nets (oWF-nets) which extend the classical WF-nets by integrating interface places to ensure asynchronous communication with the different partners. Hence, a composition is modeled by a set of oWF-nets interacting via interface places. Note that these places are used to connect only transitions from different processes.

Now when we focus on incorporating time constraints, we find different proposals which are based on extensions of Petri nets. In general, the time constraints are modeled either by durations or delays. When they are specified by durations (constants), the extension associated is said to be *Timed Petri nets*. These constant durations are either attached to places (the subclass is known as P-Timed Petri nets) or to transitions (T-Timed Petri nets). In case of delays adoption, the constraints are specified by means of intervals specifying the minimum and the

maximum amounts of time representing the transitions firing delays, the associated extension is called Time Petri nets. These intervals are attached to places (P-Time Petri nets), transitions (T-Time Petri nets) or arcs (A-Time Petri nets) leading thus to different extensions with variant semantics.

In the case of workflow processes, several studies (van der Aalst (1993); Atluri and Huang (1996); Ling and Schmidt (2000); MarteGou et al. (2001)) have shown the importance of temporal reasoning in the specification of workflow systems. In (Ling and Schmidt (2000)), the authors extend the simple WF-net presented by van der Aalst (van der Aalst (1997)) by associating with each transition a time amount representing the task duration. A temporal extension of the WF-net, called Timed WF-net is proposed. The semantic adopted here is that each enabled transition must fire immediately, otherwise the transition will be disabled. In case of firing, its duration should be respected, i.e. this transition can not be delayed. Although this approach is simple, it is strict in the sense that it requires fixed durations. To deal with this limitation, Time Workflow nets (TWF-nets) are proposed allowing to associate time delays with the activities by incorporating to each transition a time interval specifying its firing delay (Boucheneb and Barkaoui (2012); Camerzan (2007); Ling and Schmidt (2000)).

Since this time consideration is flexible and given that we are interested by modeling the composition of workflow processes with time constraints, we propose the Time open Workflow Net model (ToWF-net) (Sbaï et al. (2014)). This model associates to each transition a static time interval which refers to the execution time or delay of the corresponding activity. The formal definition of a ToWF-net model is the following:

Definition 1 ***(ToWF-net)***

A Time open Workflow net N is a tuple (P, T, F, FI, I, O) with:

- *P is a set of places,*
- *T is a set of transitions,*
- *I is a set of places which represent input interfaces that are responsible of messages receiving from other processes: $^\bullet I = \emptyset$.*
- *O is a set of places which represent output interfaces responsible of messages sending to other partners: $O^\bullet = \emptyset$.*
- *I, O and P are disjoint. I and O connect transitions of different processes.*
- *$F \subseteq ((P \cup I) \times T) \cup (T \times (P \cup O))$ is the flow relation,*
- *$FI : T \to \mathbb{Q}^+ \times \mathbb{Q}^+ \cup \{\infty\}$ is the function which associates to each transition $t \in T$ a static firing interval, i.e. $FI(t) = [minFI(t), maxFI(t)]$ where $minFI(t)$ and $maxFI(t)$ are rational numbers representing the minimal and the maximal firing time respectively,*

The marking of N is a vector of $\mathbb{N}^P$ such that for each place $p \in P$, $M(p)$ is the number of tokens in p. The initial marking of N is M_i knowing that M_p is used to denote a marking for which $M(p) = 1$ and $M(q) = 0 \; \forall q \in (P \cup I \cup O) \setminus \{p\}$.

A transition t is enabled in a marking M if the required tokens are present in the input places of t. We denote by $En(M)$ the set of all the transitions enabled in the marking M. A transition t is said disabled by the firing of t' in M if it is enabled in M but it isn't in $M - {}^\bullet t'$. When focusing of newly enabled transitions after firing a transition t from M and leading to M', we denote by $NEn(M, t)$ the set of transitions enabled after this firing.

$$NEn(M, t) = \{t' \in En(M') | t' = t \vee \neg M \geq^\bullet t +^\bullet t'\}.$$

When a transition t becomes enabled, its firing interval is set to its static firing interval $FI(t)$. The upper and lower bounds of $FI(t)$ decrease synchronously with time, until t is fired or disabled by another firing. t can fire, if the lower bound of its firing interval reaches 0, but when upper bound of its firing interval reaches 0, t must be fired without any additional delay (strong semantic). The firing takes no time but may lead to another marking.

Let us define first the state of a ToWF-net and then the transition relation.

Definition 2 *A state in a ToWF-net is the state of the process that is modeled with ToWF-net after the occurrence of an event. Formally, we characterize a state in a ToWF-net by a pair (M, Int) where:*

- *M is a marking,*
- *Int is a firing interval function, $Int : En(M) \to \mathbb{Q}^+ \times \mathbb{Q}^+ \cup \{\infty\}$. We denote $Int(t) = [minInt(t), maxInt(t)]$.*

The initial state of a ToWF-net is (M_0, Int_0) where $M_0 = M_i$ (since in a ToWF-net, only i contains initially one token) and $Int_0(t) = FI(t) \; \forall t \in En(M_0)$.

Starting from the initial state (M_0, Int_0), the net evolves following the occurrence of events. An event corresponds to either a transition firing or a time progression. Hence, we define the transition relation

between states $s_1 = (M_1, Int_1)$ and $s_2 = (M_2, Int_2)$ by $\stackrel{d}{\rightarrow}$ in case of time progression and by $\stackrel{t}{\rightarrow}$ in case of a firing. We explicit in the following the conditions of state evolution and how to compute the resulting state after an event occurrence.

1. $s_1 \stackrel{t}{\rightarrow} s_2$ if and only if s_2 is immediately reachable from s_1 by firing the transition t, i.e.

   $t \in En(M_1)$ and $minInt_1(t) = 0$,

   $M_2 = M_1 -^{\bullet} t + t^{\bullet}$, and

   $\forall t' \in En(M_2), \quad Int_2(t') = \begin{cases} FI(t')\, if t' \in NEn(M_1, t) \\ Int_1(t') \quad otherwise \end{cases}$

2. $s_1 \stackrel{d}{\rightarrow} s_2\ \forall d \in \mathbb{R}$ if and only if state s_2 is reachable from s_1 by a time progression with d time units, i.e.

   $minInt_1(t) + d \leq maxFI(t)$,

   $M_2 = M_1$, and

   $\forall t \in En(M_1)$, $Int_2(t) = [Max(0, minInt_1(t) - d), maxInt_1(t) - d]$

We can now define the semantics of a ToWF-net N by a transition system $(S, s_0, \rightarrow)$ with S the set of all the reachable states from the initial state s_0 by $\rightarrow$ the transition relation defined above.

We present in the following subsection some results of reachability analysis of ToWF-nets composition.

2.2. Reachability analysis of ToWF-nets

After the formal definition of ToWF-nets, we focus now on their reachability analysis. This analysis is based on the efficient construction of the state space.

By analogy with the marking graph defined in the context of an ordinary Petri net, we define a state space by a graph containing all accessible states of a ToWF-net from the initial state. Therefore, to calculate the state space of a ToWF-net, we must be able to calculate the reachable states by activating the enabled transitions.

Definition 3 *The state space of a ToWF-net has the following structure: $SS = (S, \rightarrow, s_0)$; where S is the set of nodes in form (M, Int) representing the reachable states from the initial one $s_0 = (M_i, Int_0)$; $\rightarrow$ represents the transition relation which defines the evolution from one state to another.*

$S = \{s | s_0 \stackrel{*}{\rightarrow} s\}$ is the set of reachable states of the model, and $\stackrel{*}{\rightarrow}$ is the reflexive and transitive closure of $\rightarrow$.

The reachability analysis (Boucheneb and Barkaoui (2012)) in timed models (such as time extensions of Petri nets as well as timed automata) is based in general on abstraction, which preserves reachability properties. Such an abstraction for timed models, consists in considering only one node for all states reachable from the same firing sequence while abstracting from their firing times. The grouped states, known as state classes, are then considered modulo some equivalence relation preserving properties of interest.

In return, the state class method is intended to provide a finite representation of the infinite state space of any bounded time Petri net.

Technical classes produce for a large class of time nets a finite representation of their behavior states, which allows a reachability analysis similar to that permitted for Petri nets by the technique of marking graph.

The state classes can be represented by a marking and a firing domain. Formally, a state class is a couple (M, D) where M is a marking and D is characterized by a set of atomic constraints over the firing delays of enabled transitions: $minFI(t) \leq t \leq maxFI(t)\ \ \forall t \in En(M)$.

Note that the initial class coincides with the initial state of the network. This initial class is (M_0, D_0) where $M_0 = M_i$ and D_0 corresponds to the firing domains of transitions enabled in M_0.

All states within the same node share the same untimed information and the union of their time domains is represented by a set of atomic constraints handled efficiently by means of a Difference Bound Matrix (DBM) (Ridi et al. (2012)). A DBM form a system of linear inequalities which constrain single variables $(v_1 ... v_n)$ and their differences within limits identified by coefficients b_{ij}. This is formally expressed as:

$$\begin{cases} v_i - v_j \leq b_{ij} \quad i, j \in [0..n],\ b_{ij} \in \mathbb{Q} \\ v_0 = 0 \end{cases}$$

In terms of behavior, this state classes group preserves highly the states traces, and thus the safety properties.

The computation of the state class graph is necessary at this point to perform the various reachability analysis. Among the abstractions proposed in the literature (Berthomieu and Diaz (1991); Berthomieu and Vernadat (2003); Yoneda and Ryuba (1998)), we consider here the state class graph method (Berthomieu and Diaz (1991)) for its advantage, over the others, which is the finiteness property for all bounded time Petri nets (while using some approximations).

3. QUANTITATIVE ANALYSIS OF TOWF-NETS

The analysis of the state space is very significant to the extent that it can reveal important characteristics of the modeled system, about its structure and dynamic behavior. However, for a more accurate verification, we should not be limited to this type of checking rather than other specific properties. Indeed, we focus in this section on the formal verification of compatibility properties of ToWF-nets as an extension of our results of compatibility analysis while abstracting time information (Sbaï and Barkaoui (2014); Sbaï et al. (2014)). These properties focus on the correctness of the interactions between the different partners.

We propose to adopt model checking method to verify these properties since this method permits an exhaustive verification over all the possible executions. Given a concurrent system Σ and a temporal logic formula φ, the model checking problem is to decide whether Σ satisfies φ. Hence, we have to formulate in temporal logic the properties to be verified.

3.1. Model checking TPN-TCTL

Real systems often have behaviors that depend on time. The ability to manipulate and model the temporal dimension of the events that take place in the real world is fundamental in many applications. These applications may involve banking, medical, or multimedia applications. The variety of applications motivate many recent studies that aim to integrate all the features necessary to take into account the time during verification.

TCTL (Timed Computational Tree Logic) is a timed extension of the temporal logic CTL. TCTL added to CTL a quantitative information on the delays between actions. It is built from atomic propositions, logical connectors and temporal operators (U, F, G, X, etc.). The TCTL temporal logic can be used to check the properties of a time Petri net.

The syntax of TCTL formulas is inductively defined by:

φ *::= false* | $\neg\varphi$ | $\varphi \wedge \varphi$ | *A(*φ *U*$_I$ φ*)*| *E(*φ *U*$_I$ φ*)*

where p denotes a proposition, φ denotes a formula and $I = [a, b]$ or $[a, \infty[$ with $a \in \mathbb{N}$ and $b \in \mathbb{N}$.

A and E are temporal quantifiers over the set of executions. $A\varphi$ announces that all the executions from the current state satisfy the property φ. $E\varphi$ states that from the current state, there exists an execution which satisfies φ. Finally $\varphi\ U_I\psi$ means that the property φ is true until ψ is true, and ψ will be true in the time interval I.

We can use other compositional temporal operators (Alur et al. (1993)): EF$_I$ φ = E(true U$_I$ φ) (Possibility), EG$_I$ φ = $\neg$ AF $_I$ $\neg\varphi$ (All locations along an execution), AF$_I$ φ = A(true U$_I$ φ) (Locations along all executions), AG$_I$ φ = $\neg$ EF $_I$ $\neg\varphi$ (All locations along all executions).

Semantically, TCTL formulas are interpreted on states and execution paths of a model $M = (S, V)$ where S is a transition system and V is a valuation function that associates with each state the set of atomic propositions it satisfies. (Konur et al. (2013))

To interpret a TCTL formula on an execution path, we introduce the notion of dense execution path. Let $s \in S$ be a state of S, $\pi(s)$ the set of all execution paths starting from s, and $\rho = s_0 \overset{d_0 t_0}{\rightarrow} s_1 \overset{d_1 t_1}{\rightarrow} s_2 ...$ an execution path of s. The dense path of the execution path ρ is the mapping $\hat{\rho} : R^+ \rightarrow S$ defined by: $\hat{\rho}(r) = s^i + \delta$ such that $r = \sum_{j=0}^{i-1} d_j + \delta$, $i \geq 0$ and $0 \leq \delta \leq d_i$.

The formal semantics of TCTL is given by the satisfaction relation defined as follows:

- $M, s \nvDash false$,
- $M, s \vDash \phi$ iff $\phi \in V(s)$,
- $M, s \vDash \neg\varphi$ iff $M, s \nvDash \varphi$,
- $M, s \vDash \varphi \wedge \psi$ iff $M, s \vDash \varphi$ and $M, s \vDash \psi$,
- $M, s \vDash \forall(\varphi \cup_I \psi)$ iff $\forall\rho \in \pi(s)\ \exists r \in I, M, \hat{\rho}(r) \vDash \psi$ and
  $\forall 0 \leq r' \leq r\ M, \hat{\rho}(r') \vDash \varphi$,
- $M, s \vDash \exists(\varphi \cup_I \psi)$ iff $\exists\rho \in \pi(s)\ \exists r \in I, M, \hat{\rho}(r) \vDash \psi$ and
  $\forall 0 \leq r' \leq r\ M, \hat{\rho}(r') \vDash \varphi$,

When interval I is omitted, its value is by default $[0, \infty[$.

The Time Petri net model is said to satisfy a TCTL formula φ iff $M,\ s_0 \vDash \varphi$.

The logic TCTL allows writing temporal properties with a quantification of the time. We chose this approach because it is decidable and PSPACE-complete for bounded Petri nets (Boucheneb et al. (2009)).

The authors of (Hadjidj and Boucheneb (2009)) have gone further by defining a sub-class of TCTL for time Petri nets in dense time, called TPN-TCTL. They proved the decidability of model-checking of TPN-TCTL on Petri nets and showed that its complexity is PSPACE.

Definition 4 *The temporal logic TPN-TCTL is defined inductively by:*

TPN-TCTL ::= false $| \varphi | \neg\varphi | \varphi \vee \psi | \varphi \wedge \psi | \varphi \Rightarrow \psi | E\varphi U_I \psi | A\varphi U_I \psi$

$| EG_I\varphi | AG_I\varphi | AF_I\varphi | EF_I\varphi | AG(\phi_1 \Rightarrow AF_{[0,d]}\phi_2)$.

Where φ and $\psi \in$ TPN-TCTL,

$I = [a, b]$ or $[a, b[$ with $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$.

ϕ_1 and ϕ_2 are propositions on markings.

$\forall G(\phi_1 \Rightarrow \forall F_{[0,d]}\phi_2)$ means that from the current state, any occurrence of marking ϕ_1 is followed by an occurrence of marking ϕ_2 less of d units of time later.

Romeo permits a practical implementation of the verification of properties described in TPN-TCTL. It is therefore possible to model check on the fly temporal quantitative properties. That's why we investigate in the following section the TCTL expression of the compatibility property and hence its verification in Romeo.

Before this, let us recall the notation used by Romeo to implement a TPN-TCTL property:

TPN-TCTL$_{Romeo}$ = $E(p)U[a, b](q) | A(p)U[a, b](q) | EF[a, b](p) | AF[a, b](p) | EG[a, b](p) | AG[a, b](p) | EF[a, b](p) | (p) \rightarrow [0, b](q)$.

where p, q: GMEC; U: until; E: exists; A: forall; F: eventually; G: always; $\rightarrow$: response; a: integer; b integer or inf (to denote ∞).

$GMEC = a{*}M(i)\{+, -\}b{*}M(j)\{<, <=, >, >=, =\}k \mid deadlock \mid bounded(k) \mid p\ and\ q \mid p\ or\ q \mid p \Rightarrow q \mid not\ p$.

M: keyword (marking); $deadlock$, $bounded$: keywords; i, j:place indexes; a, b, k :integers ; $*, +, -, and, or, \Rightarrow, not$: usual operators ; p, q: GMEC

The syntax $(p) \rightarrow [0, b](q)$ denotes a leads to property meaning $AG((p)$ imply $AE[0, b](q))$. E.g. $(p) \rightarrow [0, b](q)$ holds iff whenever p holds eventually q will hold as well in $[0, b]$ time units.

3.2. TCTL characterization of ToWF-nets compatibility

In a composition of two or more processes, the correctness of the composite process refers in general to the compatibility of the different processes. From a behavioral point of view, the involved processes are compatible if they can interact properly. This means that composite process does not suffer from any deadlock problem. There is a distinction between syntactic compatibility which refer to the conformance of interfaces (number of interfaces, names, input, ouptut, etc.) and semantic compatibility which refer to the absence of deadlocks in the global system. In this paper, we focus only on defining and verifying the semantic compatibility.

Before this, let us characterize the composite system obtained by superposing a number of ToWF-nets which are supposed syntactically compatible. The composite net N of nbX ToWF-nets N_1... N_{nbX} consists of all ToWF-nets sharing interface places. In this composite net, every input interface of a TWF-net has to be an output interface place of another TWF-net of N. Trivially, N form a time Petri net with nbX output places and nbX input places. The initial marking of N is $M_0 = \sum_{s=1}^{nbX} i_s$.

In (Bordeaux et al. (2005); Foster et al. (2004); Martens (2003)), the authors characterized the compatibility of non timed services as the absence of deadlock in the composite service. They considered that two or more oWF-nets are compatible if they can (all) reach their final states. By analogy, we consider that two or more ToWF-nets are compatible if they reach their final states as well as the timing constraints are respected.

In order to relax this definition, we propose different categories of the compatibility property.

- Partial compatibility: A composite net N satisfies the partial compatibility if it is deadlock-free.
- Total Compatibility: A composite net N is totally compatible if it is deadlock-free and furthermore, the overall process terminates properly.
- Perfect compatibility: A composite net N is perfectly compatible if it is totally compatible and the deadline constraints are satisfied.
- Incompatibility: A composite net N is incompatible if it is neither partially nor totally compatible.

We focus in what follows on formulating the four types of compatibility properties. Let us consider the following notation:

- nbX: the number of processes;
- nbp: the number of places in a given process;
- nbi: the number of interface places in a composition;
- i_s: the input place of the process number s.
- f_s: the output place of the process number s.

Partial compatibility

Partial compatibility of nbX processes refers to the absence of deadlock in their composition. A net is deadlock-free if and only if there is at least a transition allowed in every marking except the final one M_{fin} which refer to the marking of all the final places f_s $(s = 1..nbX)$. This property is expressed as follows:

$$\forall M \in [M_0\rangle, M_{fin} \in [M\rangle$$

Logically, this deadlock-freeness property can be expressed as follows: "for all the executions made possible from the initial state, no deadlock is encountered until the final state is reached". The final state is characterized by the marking of the final places. Then, we can express the partial compatibility by the following TCTL formula:

$$AG_{[0,\infty[}((not\ MF) \;\; \Rightarrow \;\; not\ deadlock)$$

In this formula, $deadlock$ is a proposition supposed to return true if and only if there is no enabled transition in the current state. MF is a proposition on marking M_{fin} assuring that each final place is marked with at least one token.

$$MF = \overset{nbX}{\underset{s=1}{\wedge}} M(f_s) >= 1$$

Here we check only the marking of final places.

Total compatibility

The proper termination of a process refers to check if this latter complete its execution in any case, and at the time of termination, all the places of the involved ToWF-nets are empty except the final places which must be marked with exactly one token. Hence, we have to check if there exists a marking M for which all the places are empty except the output ones. Formally, this property can be expressed as follows:

$$\forall M \in [M_0\rangle :$$
$$M(f_s) \geq 1 \; \forall s \in \{1,..,nbX\} \;\; \Rightarrow \;\; M = \sum_{s=1}^{nbX} f_s$$

We can formulate the proper termination in TCTL as follows:

$$AF_{[0,\infty[}\ StrictMF$$

With $StrictMF$ a proposition on the marking with a token in every final place f_s but no tokens in the other places including the interface ones.

$$StrictMF = \overset{nbX}{\underset{s=1}{\wedge}} \;\; (\overset{nbip}{\underset{p=1}{\wedge}} (M(p) = 0)$$
$$\wedge \;\; (M(f_s) = 1)) \quad \wedge \quad (\overset{nbi}{\underset{i=1}{\wedge}} M(I_i) = 0)$$

where $nbip$ is used to denote the number of places other than the final place in a process.

Perfect compatibility

When we have a strict overall deadline that the system should respect, we can enforce the total compatibility by adding this constraint. We define, in this sense, the perfect compatibility which refers to both deadlock-freeness and proper termination while taking into account the deadline verification.

Let us suppose that the deadline constraint is considered, this can be checked by verifying that a process has to terminate (reach its final state) in Tm time units. Which lead to the expression of the proper termination within this delay as follows:

$$AF_{[0,Tm]}\ StrictMF$$

Hence, these two TCTL formulas can be used to express the perfect compatibility of ToWF-nets composition:

- $AG_{[0,Tm]}((not\ MF) \;\; \Rightarrow \;\; not\ deadlock)$
- $AF_{[0,Tm]}\ StrictMF$

Incompatibility

The incompatibility is a situation in which neither deadlock-freeness nor proper termination is verified. In other words, a set of ToWF-nets are incompatible if all the possible executions don't lead to final states of the different processes. We may distinguish here between strong incompatibility checked in the interval $[0, \infty[$ and weak incompatibility which refers to incompatibility in a limited interval $[a, b]$.

If we consider that the interval $[x, y]$ may correspond to $[0, \infty[$ or $[a, b]$, the expression of the incompatibility in TCTL leads to the following formula:

$$AG_{[x,y]}((not\ MF))$$

3.3. Model checking ToWF-nets composition

We present in this subsection some results related to the analysis of the compatibility and soundness properties for a ToWF-nets composition. This verification is ensured by Romeo (Gardey et al. (2005)) which is a software studio dedicated to time Petri nets analysis. Romeo is developed by the Real-Time Systems Team at IRCCyN. It performs analysis on Transition Time Petri Nets as well as on one of their extensions dedicated to scheduling. Romeo is chosen because it assures, among other features, a State Class Graph (SCG) computation and a graphical simulation of Transition Time Petri Nets. Moreover, Romeo is used here because it

implements an on the fly model checking algorithm of TPN-TCTL formulas.

We propose to study a simple composition of ToWF-nets (figure 1). We can easily verify that no deadlock will be encountered until final places are marked. Then the partial compatibility is satisfied (see figure 2). However, the firings of transitions T_4 and T_5 of the second ToWF-net lead to two tokens in its final place f_2; this violates the total compatibility property. In the figure 3, we show the result for this property and an execution trace. After that, we propose a correction in figure 4 for which the total compatibility is verified (see figure 5).

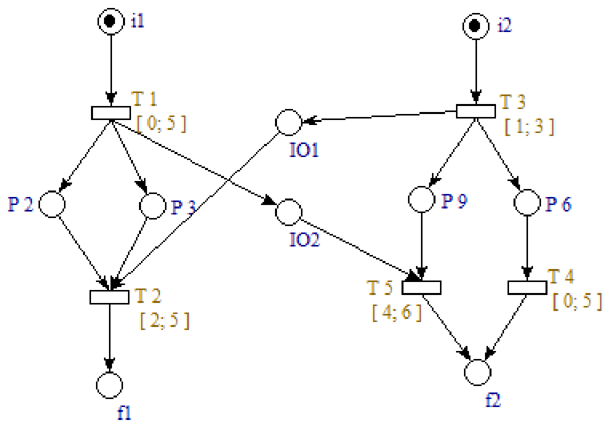


Figure 1: *Example 1 of ToWF-nets composition*

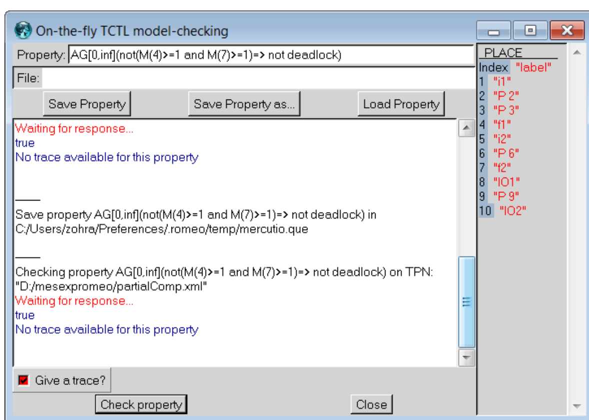


Figure 2: *Partial compatibility is verified*

4. PARAMETRIC ANALYSIS OF TOWF-NETS

In this section, we tackle with a parametric quantitative analysis of the composition of ToWF-nets. This suppose to consider k instances of each ToWF-nets ready to be executed.

For sake of covering the maximum number of real world processes, we tackle with time processes which share resource places. This places possess

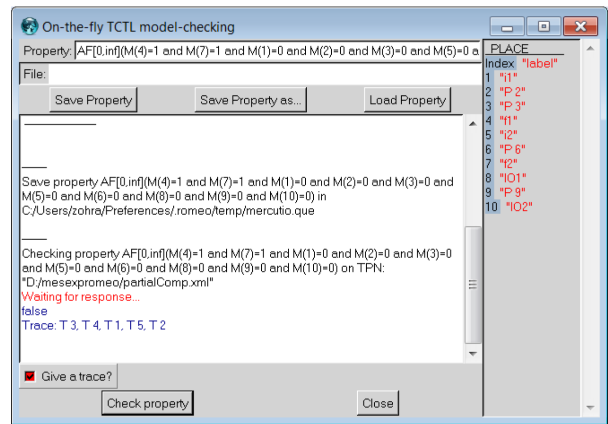


Figure 3: *Total compatibility is not verified*

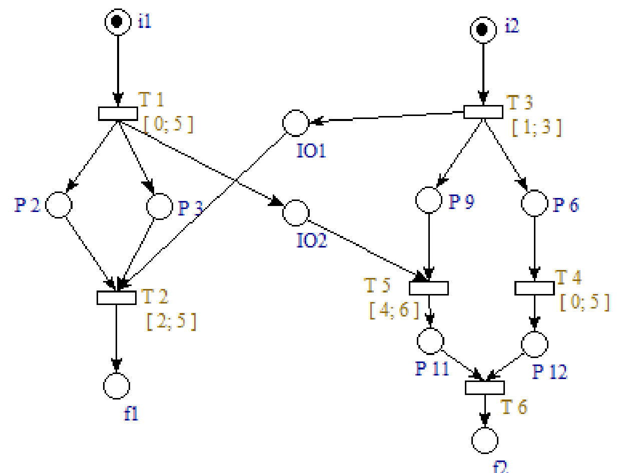


Figure 4: *Example 2 of ToWF-nets composition*

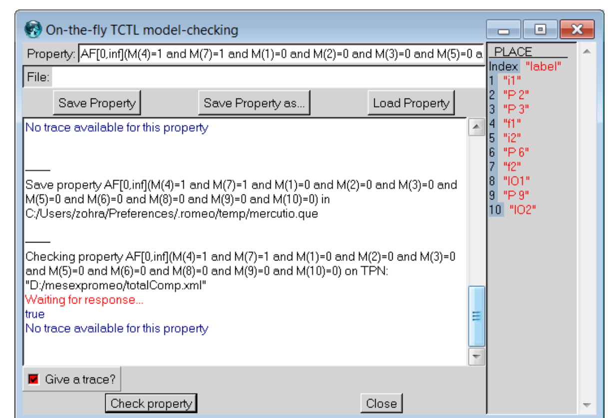


Figure 5: *Total compatibility is verified*

tokens in the beginning of each process and these tokens will be used and released after their use, i.e. at the end of execution, the resource places will regain their same initial markings.

For this, we define the time open WF-nets with shared resources (ToWFR-nets) which stand for a

class of Petri nets dedicated to model workflow processes with: time delays, resource places as well as interface places.

Definition 5 ***(ToWFR-net)***

A ToWFR-net N is a tuple $(P, RP, T, F, RF, FI, I, O, W_{RP}, M_0)$ with:

- *(P, T, F, FI, I, O) is a ToWF-net,*
- *RP is a set of resource places,*
- *I, O, P and RP are disjoint,*
- *$RF \subseteq (RP \times T) \cup (T \times RP)$ is the flow relation for resources,*
- *$W_{RP} : RF \mapsto \mathbb{N}$ is the weight function defining the weight of arcs linking the resource places. To ensure the use of resource places, we require: $W_{RP}(u) \geq 1 \forall u \in RF$*
- *M_0 is the initial marking: $M_0 = k.i + M_{RP}$ where k is the number of tokens in the initial place i and M_{RP} is the marking of resource places.*

For the interaction of ToWFR-nets, we use only interface places; resource places are used to model resources sharing between activities of the same process.

In the sequel, we will use these notations:

- IO_j refers to interface place number j
- RP_j refers to resource place number j.

As an example of ToWFR-nets composition, we study a manufacturing lane of three machines which collaborate together to manufacture some product. The composite net describing the manufacturing workflow is given in figure 6.

In this example, each machine will be launched twice (2 tokens in each initial place i_j) and share an internal resource modeled by RP_j with j the number of machines. The three machines communicate via the two interface places IO_1 and IO_2.

Now, to verify if the interacting ToWF-nets communicate correctly and if their collaboration process is sound, we propose to extend the above classes of compatibility properties with the consideration of the concurrent instances ready for execution.

4.1. K-compatibility analysis

We define the following formulas: the k-partial compatibility and the k-total compatibility.

k-partial compatibility

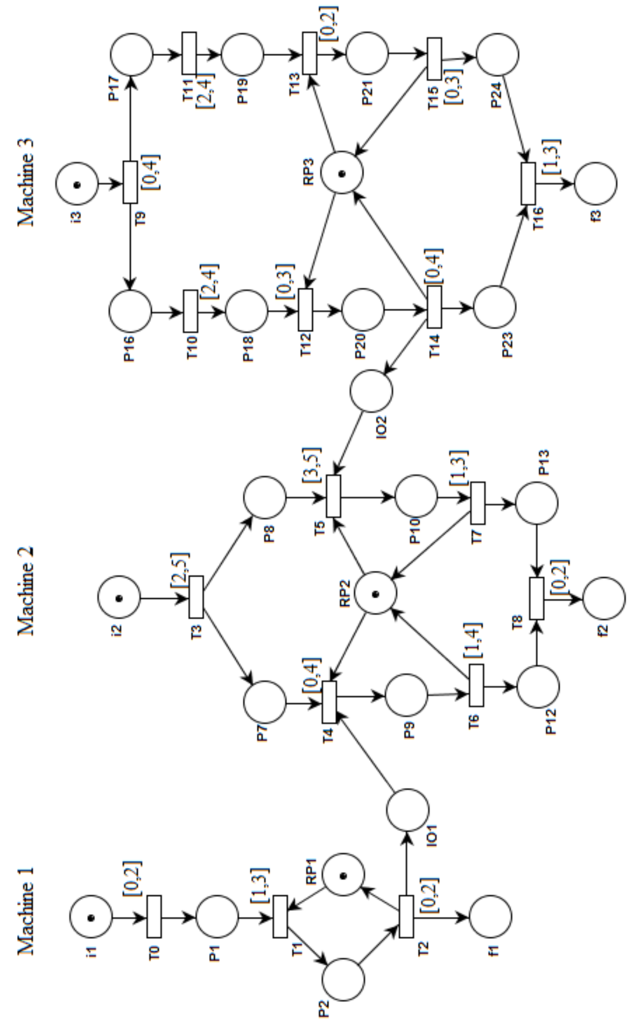


Figure 6: *Composition of three machines' processes*

The k-partial compatibility refers to verify if all the instances of the involved processes terminate without encountering a deadlock. This is possible by extending the deadlock-freeness specified in the previous section with a test of the termination of the k instances. This can be specified by the following TCTL formula:

$$AG[0, \infty[(not(\overset{nbX}{\underset{s=1}{\wedge}}(M(index_of_f_s) >= k)) \;=>\; not\ deadlock)$$

Where nbX is the number of involved processes

For the three machines example (6) this formula is written in Romeo as follows:

$$AG[0, inf](not(M(4) >= 2\ and\ M(7) >= 2\ and\ M(26) >= 2) \;=>\; not\ deadlock)$$

Where 4,7 and 26 are the indexes attributed by Romeo to the three final places.

The figure 7 shows that the two-partial compatibility is guaranteed for the three machines example.

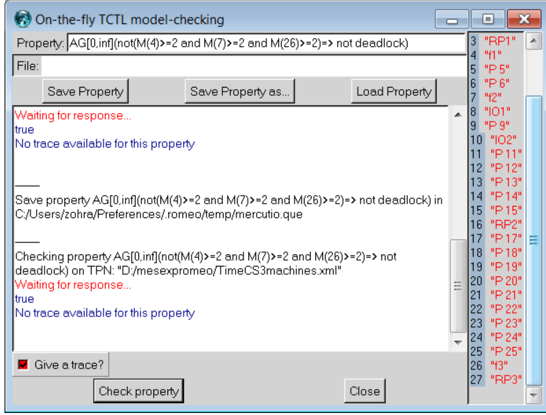


Figure 7: *Analysis example of the two-partial compatibility*

k-total compatibility

The k-total compatibility refers to verify if all the instances of the involved processes terminate properly. This means that we have to verify if we reach the state in which final places have k tokens and resource places regain exactly their initial marking and all the other places are empty. This can be specified by the following TCTL formula:

$$AF[0,\infty[(\bigwedge_{s=1}^{nbX}(M(index_of_f_s) = k) \wedge \bigwedge_{j=1}^{nbR}(M(index_of_RP_j) = m_{RP_j}) \wedge \bigwedge_{j=1}^{nbop}(M(index_of_P_j) = 0))$$

Where nbX refers to the number of interacting processes, nbR refers to the number of resource places, m_{RP_j} refers to the initial marking of resource place RP_j and $nbop$ refers to the number of places which are neither final places nor resource places.

For the three machines example (6) this formula is written in Romeo as follows:

$$AF[0, inf](M(4) = 1 \ and \ M(7) = 1 \ and \ M(26) = 1 \ and \ M(27) = 1 \ and \ M(16) = 1 \ and \ M(3) = 1 \ and \ M(1) = \ and \ M(2) = 0 \ and \ M(5) = 0 \ and \ M(6) = 0 \ and \ M(8) = 0 \ and \ M(9) = 0 \ and \ M(10) = 0 \ and \ M(11) = 0 \ and \ M(12) = 0 \ and \ M(13) = 0 \ and \ M(14) = 0 \ and \ M(15) = 0 \ and \ M(17) = 0 \ and \ M(18) = 0 \ and \ M(19) = 0 \ and \ M(20) = 0 \ and \ M(21) = 0 \ and \ M(22) = 0 \ and \ M(23) = 0 \ and \ M(24) = 0 \ and \ M(25) = 0)$$

The figure 8 shows that the two-total compatibility is guaranteed for the three machines example.

After these tests, we highlight some results on the characterization of quantitative properties of

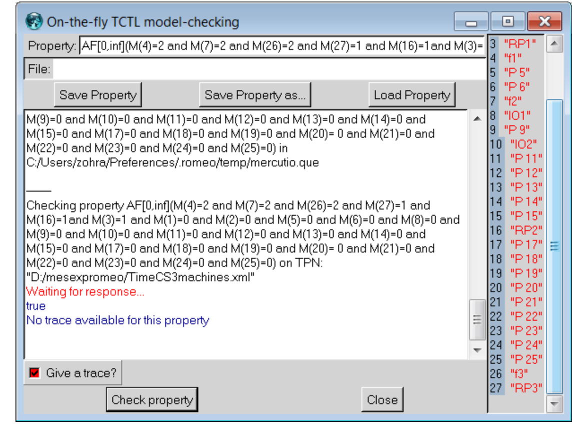


Figure 8: *Analysis example of the two-total compatibility*

ToWF-nets composition. Let us consider $\mathcal{N}$ the time workflow net obtained by composing the involved ToWF-nets and the adding of two special places p_{start} and p_{end} and two transitions t_{start} and t_{end} which connect respectively p_{start} to the the input places of the different ToWF-nets and the different final places to p_{end}.

We can easily prove that if the involved ToWF-nets are totally compatible then $\mathcal{N}$ is sound. In addition if the ToWF-nets are k-totally compatible then $\mathcal{N}$ is k-sound. But the reverse is incorrect, i.e. $\mathcal{N}$ is k-sound $\nrightarrow$ the ToWF-nets are k-totally compatible.

5. RELATED WORK

Several works dealt with the analysis of compatibility properties of processes modeled by open workflow nets or by other formalisms. Wil M. P. van der Aalst and al. (van der Aalst et al. (1998)) showed that two or more processes are compatible if their interfaces are compatible and there are no deadlocks. In addition, other concepts were formulated in relation with compatibility such as strategy and controllability.

Marlon Dumas and al. (Dumas et al. (2008)) studied the incompatibility of Web services and classified it into two types such that the incompatibility of signatures and the protocol incompatibility. The former occurs when a service requests an operation which is not possible from another service. The latter occurs when a service enters in a series of interactions with an other service, but there is no compatibility in the two services orders.

Lucas Bordeaux and al. (Bordeaux et al. (2005)) tackled the verification of Web services compatibility while assuming that not only the exchanged messages are semantically of the same type but also they have the same name. For the modeling of Web services, their work is based on labeled transition systems (LTS). The authors defined three

types of compatibility: absence of deadlock, opposite behavior and unspecified reception.

Wei Tan and al. (Tan et al. (2009)) proposed an approach that checks interface compatibility of Web services described by BPEL, and corrects these services if they are not compatible. To do this, they modeled the composition by SWF-nets, a subclass of CPN (Colored Petri Nets). Then they checked the compatibility of interfaces.

While these approaches dealt with non time processes, we focused on those constrained by time information. To check the compatibility of interacting processes, we chose to extend oWF-nets with delays associated to activities. In addition, we were based on the formal verification in our approach. We mainly used the model checking formal method to check the compatibility classes of ToWF-nets, which shows a counter example in case a property is violated allowing thus to recognize and correct the eventual errors as early as possible.

6. CONCLUSION

Nowadays, technological progress plays a fundamental role in the optimization of production processes in different sectors, especially facing the varieties produced and the constraints of productivity, capability, quality and competitiveness. For this, a prior verification of such processes and their interaction is tremendous.

We proposed first in this paper a model for processes interaction based on Petri nets. Then, we studied the compatibility of these processes by model checking techniques. In particular, we applied a TCTL model checking of these properties and simulated some examples on the Romeo model checker. Finally, we enhanced these results by taking into account several instances ready for execution as well as a possibility of sharing resources. Hence, we introduced to parametric verification of interacting processes. In future, we propose to strengthen this parametric analysis of ToWF-nets and ToWFR-nets.

REFERENCES

Alur, R. and Courchoubetis, C. and Dill, D. (1993) *Model checking in dense real time*. Information and computation (104). pp 2-34.

Atluri, V. and Huang, W. (1996) *An authorization model for workflows*. Proceedings of the 4th European Symposium on Research in Computer Security, London, Springer-Verlag. pp 44-64.

Barkaoui, K. and Ben Ayed, R. and Sbaï, Z. (2007) *Workflow Soundness Verification based on Structure Theory of Petri Nets*. International Journal of Computing and Information Sciences (IJCIS), pp. 51-61.

Berthomieu, B. and Diaz, M. (1991) *Modeling and verification of time dependent systems using time Petri nets*. IEEE Transactions on Software Engineering, 17(3).

Bordeaux, L. and Salaun, G. and Berardi, D. and Mecella, M. (2005) *When are two web services compatible ?*. Sapienza University, 3324.

Boucheneb, H. and Barkaoui, K. (2013) *Reducing interleaving semantics redundancy in reachability analysis of time Petri nets*. ACM Transactions in Embedded Computing Systems (TECS), 12(1), pp 1-24.

Boucheneb, H. and Barkaoui, K. (2012) *Parametric Verification of Time Workflow Nets*. 24th International Conference on Software Engineering (SEKE), pp 375-380.

Berthomieu B. and F. Vernadat F. (2003) *State class constructions for branching analysis of time Petri nets*. TACAS 2003, volume 2619 of Lecture Notes in Computer Science, pp 442-457.

Dumas, M. and Benatallah, B. and Motahari Nezhad, H. (2008) *Web Service Protocols : Compatibility and Adaptation*. Institute of Electrical and Electronics Engineers.

Guermouche, N. and Perrin, O. and Ringeissen, C. (2008) *Timed Specification For Web Services Compatibility Analysis*. Theoretical Computer Science.

Hadjidj, R. and Boucheneb, H. (2009) *On-the-Fly TCTL Model-Checking for Time Petri Nets*. Theoretical Computer Science, 410(42), pp 4241-4261.

Camerzan, I. (2007) *On Soundness for Time Workflow Nets*. Computer Science Journal of Moldova, 15(1), pp 74-87.

De Michelis, G. and Ellis, C. and Memmi, G. (1994) *Proceedings of the second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms, Zaragoza, Spain.*.

Foster, H. and Uchitel, S. and Magee, J. and Kramer, J. (2004) *Compatibility Verification for Web Service Choreography*. Proceedings of IEEE International Conference on Web Services, pp 738-741.

Ellis, C. and Keddara, K. and Rozenberg, G. (1995) *Dynamic change within workflow systems*. Proceedings of conference on Organizational computing systems, pp 10-21.

Esparza, Javier and Silva, Manuel (1989) *Circuits, handles, bridges and nets*. Applications and Theory of Petri Nets, Lecture Notes in Computer Science, 483, pp 210-242.

Martens, A. (2003) *On Compatibility of Web Services*. Petri Net Newsletter, pp 12-20.

Gardey, G. and Lime, D. and Magnin, M. and Roux, O.H. (2005) *Romeo: A Tool for Time Petri Nets Analysis*. Proceeding of 17th International Conference on Computer Aided Verfication (CAV'05), volume 3576 of Lecture Notes in Computer Science, pp 418-423.

Gou, H. and Huang, B. and Liu, W. and Li, Y. and Ren, S. (2001) *Modeling distributed business processes of virtual enterprises based on the object-oriented approach and petri nets*. Systems Man and Cybernetics.

Konur, S. and Fisher, M. and Schewe, S. (2013) *Combined model checking for temporal, probabilistic, and real-time logics*. Theoretical Computer Science, 503, pp 61-88.

Ling, S. and Schmidt, H. (2000) *Time Petri Nets for Workflow Modelling and Analysis*. IEEE International Conference on Systems, Man, and Cybernetics, pp 3039-3044.

Martens, A. (2005) *Analyzing Web service based business processes*. Proceeding of International Conference on Fundamental Approaches to Software Engineering, Part of the European Joint Conferences on Theory and Practice of Software, Lecture Notes in Computer Science vol. 3442.

Massuthe P. and Reisig W. and Schmidt K. (2005) *An Operating Guideline Approach to the SOA*. Annals of Mathematics, Computing and Teleinformatics, pp 35-43.

Ridi, L. and Torrini, J. and Vicario, E. (2012) *Developing a Scheduler with Difference-Bound Matrices and the Floyd-Warshall Algorithm*. IEEE SOFTWARE.

Sbaï, Z. and Barkaoui, K. (2013) *Vérification formelle des processus workflow - Extension aux workflows inter-organisationnels*. Revue Ingénierie des Systèmes d'Information: Ingénierie des systèmes collaboratifs, 18(5), pp 33-57.

Sbaï, Z. and Barkaoui, K. (2012) *Vérification Formelle des Processus Workflow Collaboratifs*. Actes de la conférence francophone sur les Systèmes Collaboratifs (SysCo'12), pp. 197-210.

Boucheneb, H. and Gardey, G. and Roux, O.H. (2009) *TCTL model-checking of Time Petri Nets*. Journal of Logic and Computation, 19(6), pp 1509-1540.

Tan, Wei and Fan, Yushun and Zhou, MengChu (2009) *A Petri Net-Based Method for Compatibility Analysis and Composition of Web Services in Business Process Execution Language*. IEEE T. Automation Science and Engineering, 6(1), pp 94-106.

Yoneda, T. and Ryuba, H. (1998) *CTL model checking of time Petri nets using geometric regions*. IEICE Transactions on Information and Systems, pp 297-396.

van der Aalst, W.M.P. and Arjan, J.M. and Christian, S. and Wolf, K. (1998) *Service Interaction: Patterns, Formalization, and Analysis*. 9th Internatinal School on Formal Methods for the design of Computer, Communication and Software Systems.

van der Aalst, W.M.P. (1997) *Verification of Workflow nets*. ICATPN 97, LNCS, 1248.

van der Aalst, W.M.P. (1996) *Three good reasons for using a petri-net-based workflow management system*. International Working Conference on Information and Process Integration in Enterprises (IPIC96), pp 179-201.

van der Aalst, W.M.P. (1993) *Interval timed coloured petri nets and their analysis*. Proceedings of the 14th International Conference on Application and Theory of Petri Nets, London, Springer-Verlag, pp 453-472.

Karsten and Schmidt (2005) *Controllability of open workflow nets*. EMISA. LNI, Bonner Köllen Verlag, pp 236-249.

Sbaï. Z. and Kamel Barkaoui. and Hanifa Boucheneb. (2014) *Compatibility Analysis of Time Open Workflow Nets*. Proceedings of the International Workshop on Petri Nets and Software Engineering, co-located with 35th International Conference on Application and Theory of Petri Nets and Concurrency (PetriNets 2014) and 14th International Conference on Application of Concurrency to System Design (ACSD), pp 249-268.

Sbaï. Z. and Kamel Barkaoui. (2014) *On Compatibility Analysis of Inter Organizational Business Processes*. Enterprise and Organizational Modeling and Simulation - 10th International Workshop, EOMAS 2014, Held at CAiSE 2014, Thessaloniki, Greece, June 16-17, 2014, Selected Papers, pp 171-186.

Verif cation of Real-Time Bounded Distributed Systems With Mobility

Bogdan Aman and Gabriel Ciobanu
Romanian Academy, Institute of Computer Science, Iaşi, Romania
bogdan.aman@gmail.com and gabriel@info.uaic.ro

We introduce and study a prototyping language for describing real-time distributed systems. Its time constraints are expressed as bounded intervals to model the uncertainty of the delay in migration and communication of agents placed in the locations of a distributed system. We provide the operational semantics, and illustrate the new language by a detailed example for which we use software tools for analyzing its temporal properties.

1. INTRODUCTION

Computer systems today are interconnected into large distributed systems. Distributed and concurrent systems are now used in both academic and industrial computing, forcing researchers and practitioners to look for theoretical models and software tools ref ecting the new framework based on mobility and interaction. Programming paradigms have progressed, allowing programmers to implement software in terms of high level abstractions. In distributed systems, such implementations are given by taking care of time scheduling, access to resources, and interaction among processes. When solving problems in distributed systems, it is useful to have an explicit notion of location, explicit migration, local interaction/communication and resource management.

Aiming to bridge the gap between the existing theoretical approach of process calculi and forthcoming realistic programming languages for distributed systems, we have introduced and studied a rather simple and expressive formalism called TIMO as a simplif ed version of timed distributed pi-calculus Ciobanu and Prisacariu (2006). In several aspects, TIMO is a prototyping language for multi-agent systems, featuring mobility and local interaction. The mobility refers to the fact that agents are in locations and that they can migrate from one location to the another, while agents must be in the same location in order to communicate. In this paper we present a revised/improved version of a real-time variant called rTIMO (Real Timed Mobility) in which the timing constraints are expressed as intervals in order to model the uncertainty of the delay in migration and communication. rTIMO supports explicit migration and local communication, together with certain timing constraints over these actions. We provide a relationship between rTIMO and the model checker UPPAAL, and so making possible formal verif cation for rTIMO. For the complex distributed systems described by such a language, we show how it is possible to use UPPAAL capabilities in order to verify certain properties.

The paper is organized as follows: Section 2 presents the syntax and semantics of rTIMO using interval constraints. Section 3 provides an example, while Sections 5 and 6 contain the modelling and verif cation in UPPAAL. Finally, Section 7 presents the related work and concludes the paper.

2. SYNTAX AND SEMANTICS OF RTIMO

The prototyping language rTIMO provides suff cient expressiveness to model in an elegant way the migration and communication in real-time distributed systems. In this paper we present a revised/improved version of rTIMO involving global timing constraints in which the timing constraints are expressed as intervals in order to model the uncertainty of the delay in migration and communication. This realistic approach is used to provide systems a larger degree of non-determinism, for instance in deciding when a process is allowed to move from one location to another one. We achieve this by assuming that a rTIMO migration process can move to another location during a time interval (and not necessarily after exactly a given number of

Processes			
P, Q	$::=$	$a^{[t_1,t_2]}!\langle v \rangle$ then P else Q |	(output)
		$a^{[t_1,t_2]}?(u)$ then P else Q |	(input)
		$\text{go}^{[t_1,t_2]} l$ then P |	(move)
		0 |	(termination)
		$id(v)$ |	(recursion)
		$P \mid Q$	(parallel)
Located Processes			
L	$::=$	$l[[P]]$	
Systems			
N	$::=$	L | $L \mid N$	

Table 1: rTIMO *Syntax*

time units). Two processes may communicate only if they are present at the same location, they use the same channel and the time constraints allow them to interact.

In rTIMO, the transitions caused by performing actions with timeouts (migration and communication) are alternated with continuous transitions (time-passing). The semantics of rTIMO is provided by multiset labelled transitions in which multisets of actions are executed in parallel (in one step).

2.1. Syntax of rTIMO

Timing constraints expressed as intervals allow to model the uncertainty of the delay in migration and communication. The syntax of rTIMO is given in Table 1, where the following is assumed:

- a set *Loc* of locations l, a set *Chan* of communication channels a, and a set *Id* of process identifiers (each $id \in Id$ has its arity m_{id});
- for each $id \in Id$ there is a unique process definition $id(u_1, \ldots, u_{m_{id}}) \stackrel{def}{=} P_{id}$, where the distinct variables u_i are parameters;
- $[t_1, t_2]$, where $t_1, t_2 \in \mathbb{R}_+$ and $t_1 \leq t_2$ is an execution time *interval* of an action;
- u is a tuple of variables;
- v is a tuple of expressions built from values, variables and allowed operations.
- a time interval $^{[t_1,t_2]}$ associated with a migration action such as $\text{go}^{[t_1,t_2]} loc$ then P indicates that process P can move to location loc after t time units, where $t \in [t_1, t_2]$.
- a time interval $^{[t_1,t_2]}$ associated with an output communication process $a^{[t_1,t_2]}!\langle z \rangle$ then P else Q makes the channel a available for communication (by sending z) for a period of $t_2 - t_1$ time units, but only after an idling of t_1 time units. It is also possible to impose an interval for an input communication process $a^{[t_1,t_2]}?(x)$ then P else Q along a channel a. In both cases, if the interaction does not happen in the interval $[t_1, t_2]$, the process gives up and continues as the alternative process Q.

The only variable binding constructor is $a^{[t_1,t_2]}?(u)$ then P else Q; it binds the variable u within P, but *not* within Q. The free variables of a process P are denoted by $fv(P)$; for a process definition, it is assumed that $fv(P_{id}) \subseteq \{u_1, \ldots, u_{m_{id}}\}$, where u_i are the process parameters. Processes are defined up-to an alpha-conversion, and $\{v/u, \ldots\}P$ denotes P in which all free occurrences of the variable u are replaced by v, eventually after alpha-converting P in order to avoid clashes.

Since location l, provided by a process $\text{go}^{[t_1,t_2]} l$ then P, can be a variable, its value can be assigned dynamically through communication with other processes; this means that migration supports a flexible scheme for the movement of processes from one location to another. Thus, the behaviour can be adapted to various changes of the distributed environment. Processes are further constructed from the (terminated) process 0, and parallel composition $P \mid Q$. A located process $l[[P]]$ specifies a process P running at location l, and a system N is built from its components L. A system N is well-formed if there are no free variables in N.

Remark 1 *In order to focus on the mobility and local interaction aspects of* rTIMO*, we abstract from arithmetical operations, considering by default that the simple ones (comparing, addition, subtraction) are included in the language.*

2.2. Operational Semantics of rTIMO

The first component of the operational semantics of rTIMO is the structural equivalence $\equiv$ over

systems. The structural equivalence is the smallest congruence such that the equalities of Table 2 hold.

~~(NNULL)~~	~~$N \mid l[[0]] \equiv N$~~
(NCOMM)	$N \mid N' \equiv N' \mid N$
(NASSOC)	$(N \mid N') \mid N'' \equiv N \mid (N' \mid N'')$
(NSPLIT)	$l[[P \mid Q]] \equiv l[[P]] \mid l[[Q]]$

Table 2: rTiMo *Structural Congruence*

Essentially, the role of $\equiv$ is to rearrange a system in order to apply the rules of the operational semantics given in Table 3. Using the equalities of Table 2, a given system N can always be transformed into a finite parallel composition of located processes of the form $l_1[[P_1]] \mid \ldots \mid l_n[[P_n]]$ such that no process P_i has the parallel composition operator at its topmost level. Each located process $l_i[[P_i]]$ is called a component of N, and the whole expression $l_1[[P_1]] \mid \ldots \mid l_n[[P_n]]$ is called a *component decomposition* of the system N.

The operational semantics rules of rTiMo are presented in Table 3. The multiset labelled transitions of form $N \xrightarrow{\Lambda} N'$ use a multiset Λ to indicate the actions executed in parallel in one step. When the multiset Λ contains only one action λ, in order to simplify the notation, $N \xrightarrow{\{\lambda\}} N'$ is simply written as $N \xrightarrow{\lambda} N'$. The transitions of form $N \stackrel{t}{\rightsquigarrow} N'$ represent a time step of length $t \in \mathbb{R}_+$.

In rule (Move0), the process $\mathsf{go}^{[0,t]} l'$ then P migrates from location l to location l' (illustrated by the label $l \rhd l'$ of the transition) and then evolves as process P. In rule (Com), an output process $a^{[0,t]}!\langle v\rangle$ then P else Q located at location l, succeeds in sending a tuple of values v over channel a to process $a^{[0,t']}?(u)$ then P' else Q' also located at l. Both processes continue to execute at location l, the first one as P and the second one as $\{v/u\}P'$. The label $\{v/u\}@l$ of the rule (Com) illustrates the fact that a communication that lead to the replacement of u by v (denoted by $\{v/u\}$) took place at location l (denoted by $@l$). If a communication action has a timer equal to $[0, 0]$, then by using the rule (Put0) for output action or the rule (Get0) for input action, the generic process $a^{[0,0]} *$ then P else Q where $* \in \{!\langle v\rangle, ?(x)\}$ continues as the process Q. Rule (Call) describes the evolution of a recursion process. The rules (Equiv) and (DEquiv) are used to rearrange a system in order to apply a rule. Rule (Par) is used to compose larger systems from smaller ones by putting them in parallel, and considering the union of multisets of actions. The rules devoted to the passing of time are starting with letter D.

A computational step is captured by a derivation of the form:

$$N \xrightarrow{\Lambda} N_1 \stackrel{t}{\rightsquigarrow} N'.$$

This means that a step is a parallel execution of individual actions of Λ followed by a time step. Performing a step $N \xrightarrow{\Lambda} N_1 \stackrel{t}{\rightsquigarrow} N'$ means that N' is directly reachable from N. If there is no applicable action ($\Lambda = \emptyset$), $N \xrightarrow{\Lambda} N_1 \stackrel{t}{\rightsquigarrow} N'$ is written $N \stackrel{t}{\rightsquigarrow} N'$ to indicate (only) the time progress.

Proposition 1 *For any systems N, N' and N'', the following hold:*

1. *If $N \stackrel{t}{\rightsquigarrow} N'$ and $N \stackrel{t}{\rightsquigarrow} N''$, then $N' \equiv N''$;*
2. *$N \stackrel{(t+t')}{\rightsquigarrow} N'$ if and only if there is a N'' such that $N \stackrel{t}{\rightsquigarrow} N''$ and $N'' \stackrel{t'}{\rightsquigarrow} N'$.*

The first item of Proposition 1 states that the passage of time does not introduce any nondeterminism into the execution of a process. Moreover, if a process is able to evolve to a certain time t, then it must evolve through every time moment before t; this ensures that the process evolves continuously.

3. TRAVEL AGENCY EXAMPLE IN RTIMO

To illustrate the syntax and semantics of rTiMo, we use an example describing an understaffed travel agency, presented also in Ciobanu and Rotaru (2013). We assume that the agency has a central office (where the executives interact with agents) and six local offices (where agents interact with customers). However, due to massive layoffs, the agency has only three travel agents available, whose jobs are to communicate special travel packages (destinations and the costs of the travel) to potential customers, and two executives whose only jobs are to assign the travel agents to certain local offices of the agency each day (not necessarily the same local office each day). Also, there are two potential customers who are interested in the recommendations made by the agency, by visiting some of the local agencies (the ones that are close to their homes). We assume that the behaviours of the agency staff and of the potential customers are cyclic, and can be described as rTiMo processes.

The first agent (i.e., process $\mathrm{Agent1}$) leaves its home (i.e., the location $\mathrm{home}_{\mathrm{Agent1}}$) and goes to the central office of the agency (i.e., location office) in order to be assigned a certain local office for the current day (i.e., a location that will replace the location variable newloc). After arriving at the central office, it has to communicate with one of the executives on channel b after signing the attendance register, any time between 1 to 5 minutes (depending on

(STOP) $$l[[0]] \not\xrightarrow{\lambda}$$

(DSTOP) $$l[[0]] \overset{t}{\rightsquigarrow} l[[0]]$$

(DMOVE) $$\frac{t_2 \geq t' > 0}{l[[\mathsf{go}^{[t_1,t_2]} l' \text{ then } P]] \overset{t'}{\rightsquigarrow} l[[\mathsf{go}^{[max\{0,t_1-t'\},t_2-t']} l' \text{ then } P]]}$$

(MOVE0) $$l[[\mathsf{go}^{[0,t]} l' \text{ then } P]] \xrightarrow{l \rhd l'} l'[[P]]$$

(COM) $$l[[a^{[0,t]}!\langle v \rangle \text{ then } P \text{ else } Q \mid a^{[0,t']}?(u) \text{ then } P' \text{ else } Q']] \xrightarrow{\{v/u\}@l} l[[P \mid \{v/u\}P']]$$

(DPUT) $$\frac{t_2 \geq t' > 0}{l[[a^{[t_1,t_2]}!\langle v \rangle \text{ then } P \text{ else } Q]] \overset{t'}{\rightsquigarrow} l[[a^{[max\{0,t_1-t'\},t_2-t']}!\langle v \rangle \text{ then } P \text{ else } Q]]}$$

(PUT0) $$l[[a^{[0,0]}!\langle v \rangle \text{ then } P \text{ else } Q]] \xrightarrow{a!^{[0,0]}@l} l[[Q]]$$

(DGET) $$\frac{t_2 \geq t' > 0}{l[[a^{[t_1,t_2]}?(u) \text{ then } P \text{ else } Q]] \overset{t'}{\rightsquigarrow} l[[a^{[max\{0,t_1-t'\},t_2-t']}?(u) \text{ then } P \text{ else } Q]]}$$

(GET0) $$l[[a^{[0,0]}?(u) \text{ then } P \text{ else } Q]] \xrightarrow{a?^{[0,0]}@l} l[[Q]]$$

(DCALL) $$\frac{l[[\{v/x\}P_{id}]] \overset{t}{\rightsquigarrow} l[[P'_{id}]]}{l[[id(v)]] \overset{t}{\rightsquigarrow} l[[P'_{id}]]} \text{ where } id(v) \stackrel{def}{=} P_{id}$$

(CALL) $$\frac{l[[\{v/x\}P_{id}]] \xrightarrow{id@l} l[[P'_{id}]]}{l[[id(v)]] \xrightarrow{id@l} l[[P'_{id}]]} \text{ where } id(v) \stackrel{def}{=} P_{id}$$

(DPAR) $$\frac{N_1 \overset{t}{\rightsquigarrow} N'_1 \qquad N_2 \overset{t}{\rightsquigarrow} N'_2}{N_1 \mid N_2 \overset{t}{\rightsquigarrow} N'_1 \mid N'_2}$$

(PAR) $$\frac{N_1 \xrightarrow{\Lambda_1} N'_1 \quad N_2 \xrightarrow{\Lambda_2} N'_2}{N_1 \mid N_2 \xrightarrow{\Lambda_1 \cup \Lambda_2} N'_1 \mid N'_2}$$

(DEQUIV) $$\frac{N \equiv N' \qquad N' \overset{t}{\rightsquigarrow} N'' \qquad N'' \equiv N'''}{N \overset{t}{\rightsquigarrow} N'''}$$

(EQUIV) $$\frac{N \equiv N' \qquad N' \xrightarrow{\Lambda} N'' \qquad N'' \equiv N'''}{N \xrightarrow{\Lambda} N'''}$$

Table 3: rTiMo *Operational Semantics*

the availability of one of the executives). Since the agent can use different means of transportations, and depending on the traff c, it can take between 5 to 10 minutes for the agent to reach the central off ce of the agency (this movement is described by the action $\mathsf{go}^{[5,10]}\mathrm{office}\ \mathtt{then}\ \mathrm{P}$). The agent then moves to the given location (it can take between 3 to 5 minutes depending on the local off ce it has to reach) and advertises (over channel a) the f rst destination on the agency's list (i.e., location dest_1), in the form of a holiday pack for 100 monetary units. Finally, after selling one travel package, the agent returns home (it can take between 5 to 8 minutes depending on the local off ce it departs from). The second and the third agent (i.e., processes $\mathrm{Agent2}$ and $\mathrm{Agent3}$) are similar to the f rst, but they have different homes (i.e., the locations $\mathrm{home}_{\mathrm{Agent2}}$ and $\mathrm{home}_{\mathrm{Agent3}}$), and advertise different destinations (i.e., locations dest_2 and dest_3), in the form of holiday packs for 200 and 300 monetary units, respectively. Formally, we have:

$$
\begin{aligned}
&\mathrm{AgentX}(\mathrm{home}_{\mathrm{AgentX}} : \mathrm{Loc}) = \\
&\quad \mathsf{go}^{[5,10]}\mathrm{office}\ \mathsf{then}\ \mathrm{AgentX}(\mathrm{office} : \mathrm{Loc}) \\
&\mathrm{AgentX}(\mathrm{office} : \mathrm{Loc}) = \\
&\quad b^{[1,5]}?(\mathrm{newloc} : \mathrm{Loc}) \\
&\qquad \mathsf{then}\ (\mathsf{go}^{[3,5]}\ \mathrm{newloc} \\
&\qquad\quad \mathsf{then}\ \mathrm{AgentX}(\mathrm{newloc} : \mathrm{Loc})) \\
&\qquad \mathsf{else}\ \mathrm{AgentX}(\mathrm{office} : \mathrm{Loc}) \\
&\mathrm{AgentX}(\mathrm{office_i} : \mathrm{Loc}) = \\
&\quad a_i^{[1,20]}!\langle \mathrm{dest_X}, 100 \cdot X\rangle \\
&\qquad \mathsf{then}\ \mathsf{go}^{[1,3]}\ \mathrm{home}_{\mathrm{AgentX}} \\
&\qquad\quad \mathsf{then}\ \mathrm{AgentX}(\mathrm{home}_{\mathrm{AgentX}} : \mathrm{Loc}) \\
&\qquad \mathsf{else}\ \mathsf{go}^{[1,3]}\ \mathrm{home}_{\mathrm{AgentX}} \\
&\qquad\quad \mathsf{then}\ \mathrm{AgentX}(\mathrm{home}_{\mathrm{AgentX}} : \mathrm{Loc}),
\end{aligned}
$$

where $1 \leq i \leq 6$ and $X \in \{1, 2, 3\}$ refers to the number of the agent.

The two executives (i.e., processes $\mathrm{Executive1}$ and $\mathrm{Executive2}$) reside at the central off ce (i.e., location office), and each chooses a local off ce (i.e., in a cyclic manner, from the locations office_1, office_3, office_5, for process $\mathrm{Executive1}$, and the locations office_2, office_4, office_6 for process $\mathrm{Executive2}$) that will be assigned to the next agent that comes to the central off ce (over channel b in a period of time between 1 and 5 time units for $\mathrm{Executive1}$ and a period of time between 2 and 4 time units for $\mathrm{Executive2}$, namely after each executive resolves some off ce paperwork that make different periods for the two executives). Formally, we have:

$$
\begin{aligned}
&\mathrm{Executive1}(\mathrm{office}_1 : \mathrm{Loc}) = \\
&\quad b^{[1,5]}!\langle \mathrm{office}_1\rangle \\
&\qquad \mathsf{then}\ \mathrm{Executive1}(\mathrm{office}_3 : \mathrm{Loc}) \\
&\qquad \mathsf{else}\ \mathrm{Executive1}(\mathrm{office}_1 : \mathrm{Loc}) \\
&\mathrm{Executive1}(\mathrm{office}_3 : \mathrm{Loc}) = \\
&\quad b^{[1,5]}!\langle \mathrm{office}_3\rangle \\
&\qquad \mathsf{then}\ \mathrm{Executive1}(\mathrm{office}_5 : \mathrm{Loc}) \\
&\qquad \mathsf{else}\ \mathrm{Executive1}(\mathrm{office}_3 : \mathrm{Loc}) \\
&\mathrm{Executive1}(\mathrm{office}_5 : \mathrm{Loc}) = \\
&\quad b^{[1,5]}!\langle \mathrm{office}_5\rangle \\
&\qquad \mathsf{then}\ \mathrm{Executive1}(\mathrm{office}_1 : \mathrm{Loc}) \\
&\qquad \mathsf{else}\ \mathrm{Executive1}(\mathrm{office}_5 : \mathrm{Loc})
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Executive2}(\mathrm{office}_2 : \mathrm{Loc}) = \\
&\quad b^{[2,4]}!\langle \mathrm{office}_2\rangle \\
&\qquad \mathsf{then}\ \mathrm{Executive2}(\mathrm{office}_4 : \mathrm{Loc}) \\
&\qquad \mathsf{else}\ \mathrm{Executive2}(\mathrm{office}_2 : \mathrm{Loc}) \\
&\mathrm{Executive2}(\mathrm{office}_4 : \mathrm{Loc}) = \\
&\quad b^{[2,4]}!\langle \mathrm{office}_4\rangle \\
&\qquad \mathsf{then}\ \mathrm{Executive2}(\mathrm{office}_6 : \mathrm{Loc}) \\
&\qquad \mathsf{else}\ \mathrm{Executive2}(\mathrm{office}_4 : \mathrm{Loc}) \\
&\mathrm{Executive2}(\mathrm{office}_6 : \mathrm{Loc}) = \\
&\quad b^{[2,4]}!\langle \mathrm{office}_6\rangle \\
&\qquad \mathsf{then}\ \mathrm{Executive2}(\mathrm{office}_2 : \mathrm{Loc}) \\
&\qquad \mathsf{else}\ \mathrm{Executive2}(\mathrm{office}_6 : \mathrm{Loc})
\end{aligned}
$$

The f rst customer (i.e., process $\mathrm{Client1}$) leaves home (i.e., location $\mathrm{home}_{\mathrm{C1}}$) when he knows the agencies should be open and visits all of the three local off ces of the agency that are closest to his home (i.e., the locations office_1, office_2, and office_3),

in order, receives travel offers from the agents found at those local off ces, and chooses the cheapest travel destination. Then, he goes to the desired destination, spends a certain amount of time there, after which he returns home. The second customer (i.e., process $\mathrm{Client2}$) has the same behaviour as the f rst, except that he has a different home (i.e., location $\mathrm{home_{Client2}}$), the off ces closest to his home are locations $\mathrm{office_4}$, $\mathrm{office_5}$ and $\mathrm{office_6}$, and that he chooses the most expensive travel destination. For simplicity, we consider that both clients have the same intervals of performing similar actions. Formally, we have:

$\mathrm{Client1}(\mathrm{home_{Client1}} : \mathrm{Loc}) =$

$\quad \mathsf{go}^{[12,13]}\ \mathrm{office_1}$

$\quad\quad \mathsf{then}\ \mathrm{Client1}(\mathrm{office_1} : \mathrm{Loc})$

$\mathrm{Client1}(\mathrm{office_1} : \mathrm{Loc}) =$

$\quad a_1^{[0,4]}?(\mathrm{dest_{Client1,1}} : \mathrm{Loc}, \mathrm{cost_{Client1,1}} : \mathbb{N})$

$\quad\quad \mathsf{then}\ (\mathsf{go}^{[1,2]}\ \mathrm{office_2}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Client1}(\mathrm{office_2} : \mathrm{Loc}))$

$\quad\quad \mathsf{else}\ (\mathsf{go}^{[1,2]}\ \mathrm{office_2}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Client1}(\mathrm{office_2} : \mathrm{Loc}))$

$\mathrm{Client1}(\mathrm{office_2} : \mathrm{Loc}) =$

$\quad a_2^{[0,4]}?(\mathrm{dest_{Client1,2}} : \mathrm{Loc}, \mathrm{cost_{Client1,2}} : \mathbb{N})$

$\quad\quad \mathsf{then}\ (\mathsf{go}^{[1,3]}\ \mathrm{office_3}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Client1}(\mathrm{office_3} : \mathrm{Loc}))$

$\quad\quad \mathsf{else}\ (\mathsf{go}^{[1,3]}\ \mathrm{office_3}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Client1}(\mathrm{office_3} : \mathrm{Loc}))$

$\mathrm{Client1}(\mathrm{office_3} : \mathrm{Loc}) =$

$\quad a_3^{[0,4]}?(\mathrm{dest_{lientC1,3}} : \mathrm{Loc}, \mathrm{cost_{Client1,3}} : \mathbb{N})$

$\quad\quad \mathsf{then}\ (\mathsf{go}^{[1,5]}\ \mathrm{next_{Client1}}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Client1}(\mathrm{next_{Client1}} : \mathrm{Loc}))$

$\quad\quad \mathsf{else}\ (\mathsf{go}^{[1,5]}\ \mathrm{next_{Client1}}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Clent1}(\mathrm{next_{Client1}} : \mathrm{Loc}))$

$\mathrm{Client1}(\mathrm{dest_{Client1,i}} : \mathrm{Loc}) =$

$\quad \mathsf{go}^{[1,5]}\ \mathrm{home_{Client1}}$

$\quad\quad \mathsf{then}\ \mathrm{Client1}(\mathrm{home_{Client1}} : \mathrm{Loc})$, for $1 \leq i \leq 3$

where

$$\mathrm{next_{Client1}} = \begin{cases} \mathrm{dest_{Client1,i}} & \text{if } \mathrm{cost_{Client1,i}} = \\ & min_{j\in\{1,2,3\}}\mathrm{cost_{Client_1,j}} \in \mathbb{N} \\ \mathrm{home_{Client1}} & \text{otherwise.} \end{cases}$$

$\mathrm{Client2}(\mathrm{home_{Client2}} : \mathrm{Loc}) =$

$\quad \mathsf{go}^{[12,13]}\ \mathrm{office_4}$

$\quad\quad \mathsf{then}\ \mathrm{Client1}(\mathrm{office_4} : \mathrm{Loc})$

$\mathrm{Client2}(\mathrm{office_1} : \mathrm{Loc}) =$

$\quad a_4^{[0,4]}?(\mathrm{dest_{Client2,1}} : \mathrm{Loc}, \mathrm{cost_{Client2,1}} : \mathbb{N})$

$\quad\quad \mathsf{then}\ (\mathsf{go}^{[1,2]}\ \mathrm{office_2}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Client2}(\mathrm{office_5} : \mathrm{Loc}))$

$\quad\quad \mathsf{else}\ (\mathsf{go}^{[1,2]}\ \mathrm{office_2}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Client2}(\mathrm{office_5} : \mathrm{Loc}))$

$\mathrm{Client2}(\mathrm{office_5} : \mathrm{Loc}) =$

$\quad a_5^{[0,4]}?(\mathrm{dest_{Client2,2}} : \mathrm{Loc}, \mathrm{cost_{Client2,2}} : \mathbb{N})$

$\quad\quad \mathsf{then}\ (\mathsf{go}^{[1,3]}\ \mathrm{office_6}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Client2}(\mathrm{office_6} : \mathrm{Loc}))$

$\quad\quad \mathsf{else}\ (\mathsf{go}^{[1,3]}\ \mathrm{office_6}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Client2}(\mathrm{office_6} : \mathrm{Loc}))$

$\mathrm{Client2}(\mathrm{office_6} : \mathrm{Loc}) =$

$\quad a_6^{[0,4]}?(\mathrm{dest_{Client2,3}} : \mathrm{Loc}, \mathrm{cost_{Client2,3}} : \mathbb{N})$

$\quad\quad \mathsf{then}\ (\mathsf{go}^{[1,5]}\ \mathrm{next_{Client2}}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Client2}(\mathrm{next_{Client2}} : \mathrm{Loc}))$

$\quad\quad \mathsf{else}\ (\mathsf{go}^{[1,5]}\ \mathrm{next_{Client2}}$

$\quad\quad\quad \mathsf{then}\ \mathrm{Client2}(\mathrm{next_{Client2}} : \mathrm{Loc}))$

$\mathrm{Client2}(\mathrm{dest_{Client2,i}} : \mathrm{Loc}) =$

$\quad \mathsf{go}^{[1,5]}\ \mathrm{home_{Client2}}$

$\quad\quad \mathsf{then}\ \mathrm{Client2}(\mathrm{home_{Client2}} : \mathrm{Loc})$, for $1 \leq i \leq 3$

where

$$\mathrm{next_{Client2}} = \begin{cases} \mathrm{dest_{Client2,i}} & \text{if } \mathrm{cost_{Client2,i}} = \\ & max_{j\in\{1,2,3\}}\mathrm{cost_{Client2,j}} \in \mathbb{N} \\ \mathrm{home_{Client2}} & \text{otherwise.} \end{cases}$$

The initial state of the corresponding rTiMo network N is:

$$\begin{gathered}\mathrm{home}_{\mathrm{Agent1}}[[\mathrm{Agent1}(\mathrm{home}_{\mathrm{Agent1}})]] \mid \\ \mid \mathrm{home}_{\mathrm{Agent2}}[[\mathrm{Agent2}(\mathrm{home}_{\mathrm{Agent2}})]] \mid \\ \mid \mathrm{home}_{\mathrm{Agent3}}[[\mathrm{Agent3}(\mathrm{home}_{\mathrm{Agent3}})]] \mid \\ \mid \mathrm{office}[[\mathrm{Executive1}(\mathrm{office}_1) \mid \mathrm{Executive2}(\mathrm{office}_2)]] \mid \\ \mid \mathrm{home}_{\mathrm{Client1}}[[\mathrm{Client1}(\mathrm{home}_{\mathrm{Client1}})]] \mid \\ \mid \mathrm{home}_{\mathrm{Client2}}[[\mathrm{Client2}(\mathrm{home}_{\mathrm{Client2}})]]\end{gathered}$$

4. TIMED SAFETY AUTOMATA

Towards a necessary automated verification of complex distributed systems described by rTiMo, we provide first a relationship between rTiMo and timed safety automata Alur and Dill (1994). Then, taking into consideration the existing software tools, we relate rTiMo to UPPAAL. UPPAAL is an integrated tool environment for modelling, validation and verification of real-time systems. More details about the semantics of the input language of UPPAAL can be found at http://www.uppaal.org/. Modelling and verification of real-time systems by using UPPAAL are presented in Hessel et all (2008). Such a system is modelled as a network of several parallel timed automata. All the clocks are evaluated to real-numbers and progress synchronously. The model uses variables just as in programming languages: they are read, written, and are subject to linear expressions. A state of the system is defined by the locations of the network, the clock constraints, and the values of the variables. In any state, every automaton from the network may fire an edge (if it satisfies the restrictions) separately or synchronize with another automaton, leading to a new state.

4.1. Syntax

Assume a finite set of real-valued variables $\mathcal{C}$ ranged over by x, y, $\ldots$ standing for clocks, and a finite alphabet Σ ranged over by a, b, $\ldots$ standing for actions. A clock constraint g is a conjunctive formula of constraints of the form $x \sim m$ or $x - y \sim m$, for $x, y \in \mathcal{C}$, $\sim \in \{\leq, <, =, >, \geq\}$, and $m \in \mathbb{N}$. The set of clock constraints is denoted by $\mathcal{B}(\mathcal{C})$.

Definition 1 *A* **timed safety automaton** *$\mathcal{A}$ is a tuple $\langle N, n_0, E, I\rangle$, where*

- *N is a finite set of nodes;*
- *n_0 is the initial node;*
- *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times N$ is the set of edges;*
- *$I : N \to \mathcal{B}(\mathcal{C})$ assigns invariants to nodes.*

$n \xrightarrow{g,a,r} n'$ is a shorthand notation for $\langle n, g, a, r, n'\rangle \in E$. Node invariants are restricted to constraints of the form: $x \leq m$ or $x < m$ where $m \in \mathbb{N}$.

4.2. Networks of Timed Automata

A network of timed automata is the parallel composition $\mathcal{A}_1 \mid \ldots \mid \mathcal{A}_n$ of a set of timed automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ combined into a single system using the parallel composition operator and with all internal actions hidden. Synchronous communication inside the network is by handshake synchronisation of input and output actions. In this case, the action alphabet Σ consists of $a?$ symbols (for input actions), $a!$ symbols (for output actions), and τ symbols (for internal actions). A detailed example is found in Henzinger (1994).

A network can perform delay transitions (delay for some time), and action transitions (follow an enabled edge). An action transition is enabled if the clock assignment also satisfies all integer guards on the corresponding edges. In synchronisation transitions, the resets on the edge with an output-label are performed before the resets on the edge with an input-label. To model urgent synchronisation transitions that should be taken as soon as they are enabled (the system may not delay), a notion of urgent channels is used.

Let u, v, $\ldots$ denote clock assignments mapping $\mathcal{C}$ to non-negative reals $\mathbb{R}_+$. $g \models u$ means that the clock values u satisfy the guard g. For $d \in \mathbb{R}_+$, the clock assignment mapping all $x \in \mathcal{C}$ to $u(x) + d$ is denoted by $u + d$. Also, for $r \subseteq \mathcal{C}$, the clock assignment mapping all clocks of r to 0 and agreeing with u for the other clocks in $\mathcal{C}\backslash r$ is denoted by $[r \mapsto 0]u$. Let n_i stand for the ith element of a node vector n, and $n[n'_i/n_i]$ for the vector n with n_i being substituted with n'_i.

A network state is a pair $\langle n, u\rangle$, where n denotes a vector of current nodes of the network (one for each automaton), and u is a clock assignment storing the current values of all network clocks and integer variables.

Definition 2 *The operational semantics of a timed automaton is a transition system where states are pairs $\langle n, u\rangle$ and transitions are defined by the rules:*

- *$\langle n, u\rangle \xrightarrow{d} \langle n, u+d\rangle$ if $u \in I(n)$ and $(u+d) \in I(n)$, where $I(n) = \bigwedge I(n_i)$;*
- *$\langle n, u\rangle \xrightarrow{\tau} \langle n[n'_i/n_i], u'\rangle$ if $n_i \xrightarrow{g,\tau,r} n'_i$, $g \models u$, $u' = [r \mapsto 0]u$ and $u' \in I(n[n'_i/n_i])$;*
- *$\langle n, u\rangle \xrightarrow{\tau} \langle n[n'_i/n_i][n'_j/n_j], u'\rangle$ if there exist $i \neq j$ such that*
  1. *$n_i \xrightarrow{g_i,a?,r_i} n'_i$, $n_j \xrightarrow{g_j,a!,r_j} n'_j$, $g_i \wedge g_j \models u$,*
  2. *$u' = [r_i \mapsto 0]([r_j \mapsto 0]u)$ and $u' \in I(n[n'_i/n_i][n'_j/n_j])$.*

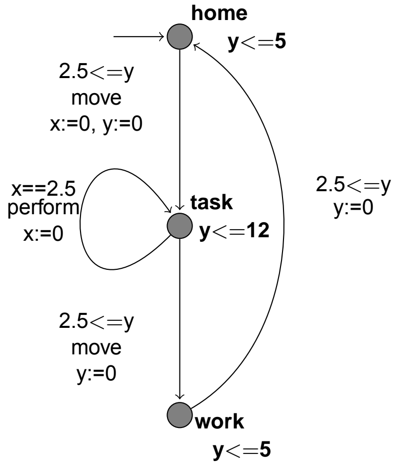


Figure 1: *Timed Safety Automata*

Graphically, a timed safety automata can be represented as a graph having a finite set of nodes and a finite set of labelled edges (representing transitions), using real-timed variables (representing the clocks of the system). The clocks are initialised with zero when the system starts, and then increased synchronously with the same rate. The behaviour of the automaton is restricted by using clock constraints, i.e. guards on edges, and local timing constraints called *node invariants* (e.g., see Figure 1). An automaton is allowed to stay in a node as long as the timing conditions of that node are satisfied. A transition can be taken when the edge guards are satisfied by clocks values. When a transition is taken, clocks may be reset to zero.

Building a timed automaton for each located process of rTiMo leads to the next result about the equivalence between an rTiMo network N and its corresponding timed automaton $\mathcal{A}_N$.

Theorem 1 *Given an* rTiMo *network* N *with channels appearing only once in output actions, there exists a network* $\mathcal{A}_N$ *of several parallel timed automata with a bisimilar behaviour.*

A bisimilar behaviour is given by:

- at the start of execution, all clocks in rTiMo and their corresponding timed automata are set to 0;
- the consumption of a go action in a node l is matched by a τ edge obtained by translation;
- a communication rule is matched by a synchronization between the edges obtained by translations;
- the passage of time is similar in both formalisms: in rTiMo the global clock is used to decrement by d all timers in the network when no action is possible, while in the timed automata all local clocks are decremented synchronously with the same value d.

Thus, the size of a timed safety automata $\mathcal{A}_N$ is polynomial with respect to the size of a TiMo network N, and the state spaces have the same number of states.

5. MODELLING THE TRAVEL AGENCY EXAMPLE IN UPPAAL

The model of the travel agency of Section 3 has three templates:

- $\mathrm{Agent(int\ dest)}$ is the model of an agent with one integer parameter dest, as shown in Figure 2. Using the parameter dest we can initialize the three agents from Section 3 by creating the processes $\mathrm{A1 = Agent(1)}$, $\mathrm{A2 = Agent(2)}$ and $\mathrm{A3 = Agent(3)}$, where each agent sells a travel package to a different destination dest. It is also possible to instantiate any number of agents, or agents that sell the same travel package.
- $\mathrm{Executive(int\ o1, int\ o2, int\ o3)}$ is the model of an executive with three integer parameters $\mathrm{o1}$, $\mathrm{o2}$ and $\mathrm{o3}$, shown in Figure 2. The three parameters are used to initiate the two executives described in Section 3 by creating the processes $\mathrm{E1 = Executive(1, 3, 5)}$ and $\mathrm{E2 = Executive(2, 4, 6)}$, where each executive is given the office locations that he/she can assign to agents. As for the agents, it is possible to create more executives than the ones presented in Section 3.
- $\mathrm{Client(int\ id, int\ o1, int\ o2, int\ o3)}$ is the model of a client with four parameters, shown in Figure 3. The parameters are used to initiate the two clients of Section 3 by creating the processes $\mathrm{C1 = Client(1, 1, 2, 3)}$ and $\mathrm{C2 = Client(2, 4, 5, 6)}$. The id parameter is used to uniquely identify a client, while the other parameters are used to identify three local offices each client is allowed to visit before making a travel decision. As for Agent and $\mathrm{Executive}$, any number of clients can be created.

Thus, the initial system is

$$\mathrm{system\ A1, A2, A3, E1, E2, C1, C2.}$$

We explain in detail the Agent template of Figure 2 (the others are constructed in a similar manner). It has five locations: home, office, $\mathrm{office_b}$, $\mathrm{office_o}$

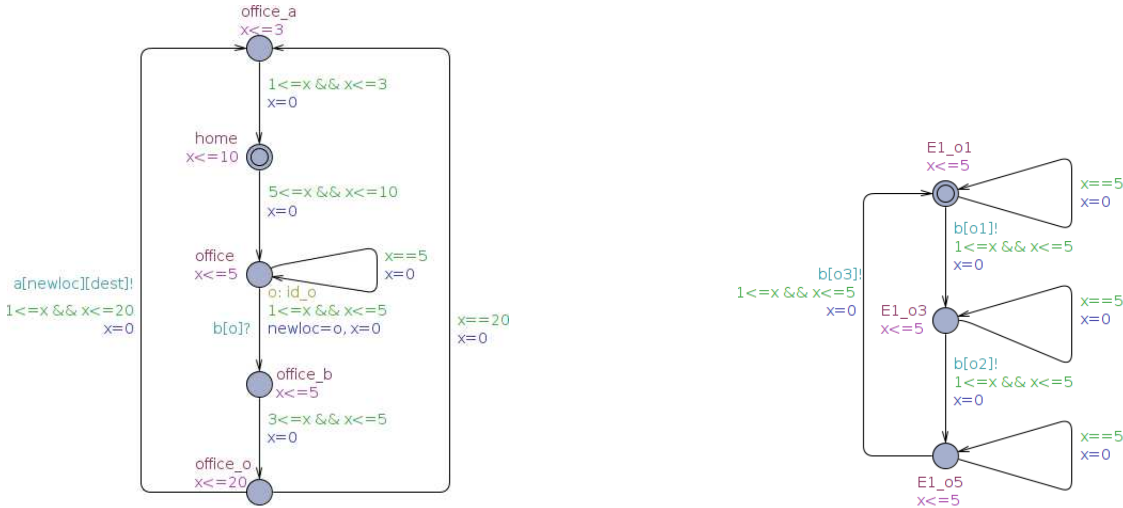


Figure 2: *The Agent (left) and Executive (right) Templates*

and $\mathrm{office_a}$. The initial location is home, which corresponds to the fact that an agent is at home. The location has the invariant $x \ <= \ 10$ (taken from the rTiMo action $\mathrm{go}^{[5,10]}$) which has the effect that the location must be left within 10 time units. The outgoing transition towards location office is guarded by the constraints $5 \ <= \ x$ and $x \ <= 10$, which correspond to the above mentioned go rTiMo action. Once at location office, the agent can either synchronize on channel $b[o]$ with an executive or, if the channel expires, create another instance in order to be able to receive an off ce location from an executive. After the communication is performed, the agent is at location $\mathrm{office_b}$ (meaning that it successfully received the location $\mathrm{newloc} = \mathrm{o}$ of the off ce he is detached to), and is ready to move to the assigned location $\mathrm{office_o}$. After arriving at $\mathrm{office_o}$, he awaits for a client for at most 20 time units, to which he must communicate the travel package on channel $\mathrm{a[newloc][dest]!}$. Regardless of the fact that he interacts with a client or not, he moves within 20 time units to the location $\mathrm{office_b}$ where he is ready to go home in order to prepare for a new working day. Thus, an agent has a cyclic evolution (a similar behaviour as one of the executives and of the clients).

6. VERIFYING PROPERTIES OF TRAVEL AGENCY BY USING UPPAAL

According to the results and descriptions presented in the previous section, we can verify time bounded distributed systems with mobility presented as rTiMo networks by using UPPAAL. UPPAAL can be used to check temporal properties of networks of timed automata, properties expressed in Computation Tree Logic (CTL). If ϕ and ψ are boolean expressions over predicates on nodes, integer variables and clock constraints, then the formulas have the following forms:

A [ ] ϕ - Invariantly ϕ; A $\langle\,\rangle$ ϕ - Always Eventually ϕ;

E [ ] ϕ - Potentially Always ϕ; E $\langle\,\rangle$ ϕ - Possibly ϕ;

$\phi \leadsto \psi$ - ϕ always leads to ψ. This is a shorthand for A [ ] ($\phi \Rightarrow$ A $\langle\,\rangle$ ψ)

The formulas can be of two types: path formulae (quantify over paths or traces of the model) and state formulae (individual states). Path formulae can be classif ed into reachability (E $\langle\,\rangle$ ϕ), safety (A [ ] ϕ and E [ ] ϕ) and liveness (A $\langle\,\rangle$ ϕ and $\phi \leadsto \psi$). Reachability properties are used to check whether there exist a path starting at the initial state, such that ϕ is eventually satisf ed along that path. Safety properties are used to verify that something bad will never happen, while liveness properties check whether something will eventually happen.

We present various properties that could be analyzed and verif ed for our example from Section 3. We have used an Intel PC with 8 GB memory, 2.50 GHz $\times$ 4 CPU and 64-bit Ubuntu 14.04 LTS to run the experiments. The results are presented for each analyzed property.

Example 1 *Given the uncertainty of the delay in migration and communication, the size of the potential interactions in* rTiMo *systems grows exponential making the software verif cation a necessity. We use* UPPAAL *to perform this kind of verif cations for the travel agency example presented in Section 3, for both safety and liveness properties. Here we present only some of the formulas/properties verif ed by using* UPPAAL.

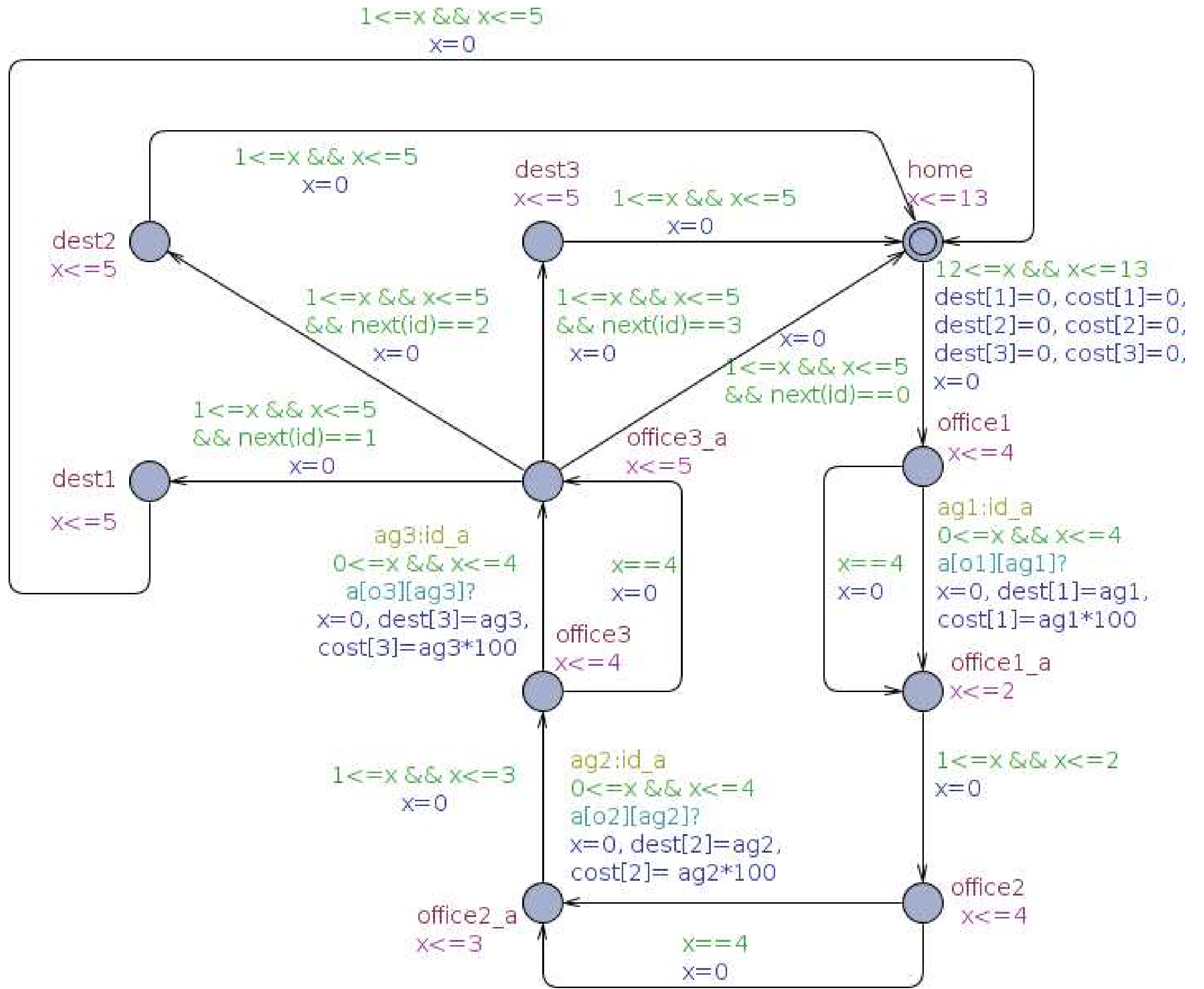


Figure 3: *The Client Template*

- C1.office1 $--\!>$ C1.home

  The formulae C1.office1 $\rightsquigarrow$ C1.home*, shorthand for* A[ ](C1.office1 $\Rightarrow$ A⟨ ⟩C1.home*, describe that, once the client* C1 *is in the* office1 *location, then it will always reach the* home *location. This implies that after leaving location* office1*, even whether the client visits or not the locations* office2 *and* office3*, the client* C1 *goes to the desired location, after which returns* home*.*

```
C1.office1 --> C1.home
Verification/kernel/elapsed time used: 46.3s / 0.66s / 47.231s.
Resident/virtual memory usage peaks: 38,620KB / 72,036KB.
Property is satisfied.
```

- E⟨ ⟩ C1.home imply C1.dest1

  This formulae is used to check whether once the client C1 *is in the* home *location, then it possibly reaches the* dest1 *location. This implies that if the client* C1 *leaves the* home *location, one of its travels takes him to location* dest1*. In a similar manner it can be checked that there are evolutions in which the client* C1 *visits one of the locations* dest2 *or* dest3*.*

```
E<> C1.home imply C1.dest1
Verification/kernel/elapsed time used: 0.07s / 0s / 0.066s.
Resident/virtual memory usage peaks: 28,148KB / 59,840KB.
Property is satisfied.
```

- E⟨ ⟩ A1.office and A2.office and A3.office

  This is checking whether or not the agents A1*,* A2 *and* A3 *reach the* office *location at the same time. Due to the uncertainty of the delay in migration and communication and of the fact that each agent has different timing constraints, having all agents at location* office *may not happen in all the possible evolutions.*

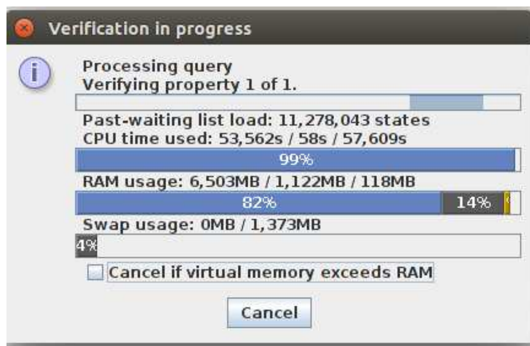
```
E<> A1.office and A2.office and A3.office
Verification/kernel/elapsed time used: 0s / 0s / 0.004s.
Resident/virtual memory usage peaks: 6,908KB / 43,516KB.
Property is satisfied.
```

- A[ ] not deadlock

  This is checking that there exists no deadlock. This implies that, whatever are the interactions between the involved participants, the evolution never stops. This means that after a working day the agents go the next day to work, while the clients continue to look for travel packages in the following days.

```
A[] not deadlock
Killing server!
Verification/kernel/elapsed time used: 53,562.62s / 58.96s / 57,609.092s.
Resident/virtual memory usage peaks: 6,861,312KB / 7,178,716KB.
Disconnected.
```

  For this property, we have stopped the verif cation process after 57609 seconds due to the fact that the RAM was fully used (as it can be seen below).



  Since the verif cation of the previous property was stopped due to insuff cient RAM, we try a similar verif cation of "no deadlock" for smaller systems. For systems with a smaller number of agents, executives and clients the "no deadlock" property is satisf ed. For one agent, one executive and one client the UPPAAL *verif cation returned:*

```
A[] not deadlock
Verification/kernel/elapsed time used: 0.12s / 0.11s / 0.236s.
Resident/virtual memory usage peaks: 5,972KB / 42,572KB.
Property is satisfied.
```

  Adding another agent takes more time to verify, but the property still holds:

```
A[] not deadlock
Verification/kernel/elapsed time used: 10.72s / 0.53s / 11.258s.
Resident/virtual memory usage peaks: 27,284KB / 59,784KB.
Property is satisfied.
```

Several other properties of rTIMO *systems can be verif ed by using* UPPAAL *.*

7. CONCLUSION AND RELATED WORK

In this paper we presented time bounded extension of rTIMO , suitable to work in complex distributed systems with mobility. It is different from all previous approaches since it encompasses specif c features as real-time timeouts given as intervals, explicit locations, time bounded migration and communication. The parallel execution of a step is provided by multiset labelled transitions. We have presented an example of applying rTIMO to an understaffed travel agency, illustrating that rTIMO provides an appropriate framework for modelling and reasoning about time bounded distributed systems with migration and interaction/communication. We have shown that we can model and verify real-time systems (e.g., the travel agency) corresponding to rTIMO networks by using UPPAAL . As rTIMO is a prototype language, a f exible representation of a travel agency is given as a number of parallel processes that are instances of the AX, EX and CX. The implementation of rTIMO processes in UPPAAL is natural due to the fact that in UPPAAL it is possible to use templates. For the running example this allows, by proper instantiations, to create any number of agents, executives and clients that can interact when placed in parallel. It is easy to note that the formalism presented in Aman and Ciobanu (2013) represents a strict subclass of the formalism presented in this paper.

Several proposals of process calculi for real-time modelling and verif cation have been presented in the literature: timed CSP Reed and Roscoe (1988), timed ACP Baeten and Bergstra (1991) and several timed extensions of CCS Moller and Tofts (1990); Yi et all (1994). Aiming to bridge the gap between the existing theoretical approach of process calculi and forthcoming realistic programming languages for distributed systems, we have introduced and studied a rather simple and expressive formalism called TIMO as a simplif ed version of timed distributed pi-calculus Ciobanu and Prisacariu (2006). In several aspects, TIMO is a prototyping language for multi-agent systems, featuring mobility and local interaction. Starting with a f rst version proposed in Ciobanu and Koutny (2008), several variants were developed during the last years; we mention here the access permissions given by a type system in perTIMO Ciobanu and Koutny (2011a), as well as a probabilistic extension pTIMO Ciobanu and Rotaru (2013). Inspired by TIMO , a f exible software platform was introduced in Ciobanu and Juravle (2009, 2012) to support the specif cation of agents allowing timed migration in a distributed environment Ciobanu (2010). Interesting properties of distributed systems described by TIMO refer to process migration, time constraints, bounded

liveness and optimal reachability Aman et. all (2012); Ciobanu and Koutny (2011b). A verif cation tool called TiMo@PAT Ciobanu and Zheng (2013) was developed by using Process Analysis Toolkit (PAT), an extensible platform for model checkers. A probabilistic temporal logic called PLTM was introduced in Ciobanu and Rotaru (2013) to verify complex properties making explicit reference to specif c locations, temporal constraints over local clocks and multisets of actions.

Acknowledgements. Many thanks to the reviewers for their useful comments. The work was supported by a grant of the Romanian National Authority for Scientif c Research, project number PN-II-ID-PCE-2011-3-0919.

REFERENCES

Alur, R. and Dill, D.L. (1994) A Theory of Timed Automata. *Theoretical Computer Science* **126**, 183–235.

Aman, B. and Ciobanu, G. (2013) Real-Time Migration Properties of rTiMo Verif ed in Uppaal. In Hierons, R., Merayo, M.and Bravetti, M. (Eds.), SEFM 2013. *Lecture Notes in Computer Science* **8137**, 31–45.

Aman, B., Ciobanu, G. and Koutny, M. (2012) Behavioural Equivalences over Migrating Processes with Timers. In Giese, H. and Rosu, G. (Eds.) FMOODS/FORTE 2012, *Lecture Notes in Computer Science* **7273**, 52–66.

Baeten, J.C.M. and Bergstra, J.A. (1991) Real Time Process Algebra. *Journal of Formal Aspects of Computing Science* **3(2)**, 142–188.

Ciobanu, G. (2008) Behaviour Equivalences in Timed Distributed π-Calculus. In Wirsing, M., Banâtre, J.-P., Hölzl, M. and Rauschmayer, A. (Eds.), *Lecture Notes in Computer Science* **5380**, 190–208.

Ciobanu, G. (2010) Finding Network Resources by Using Mobile Agents. *Intelligent Distributed Computing IV. Studies in Computational Intelligence* **315**, 305–313.

Ciobanu, G. and Juravle, C. (2009) A Software Platform for Timed Mobility and Timed Interaction. In Lee, D., Lopes, A. and Poetzsch-Heffter, A. (Eds.) FMOODS/FORTE 2009, *Lecture Notes in Computer Science* **5522**, 106–121.

Ciobanu, G. and Juravle, C. (2012) Flexible Software Architecture and Language for Mobile Agents. *Concurrency and Computation: Practice and Experience***24**, 559–571.

Ciobanu, G. and Koutny, M. (2008) Modelling and Verif cation of Timed Interaction and Migration. In Fiadeiro, J.L., Inverardi, P. (Eds.) FASE 2008, *Lecture Notes in Computer Science* **4961**, 215–229.

Ciobanu, G. and Koutny, M. (2011) Timed Migration and Interaction With Access Permissions. In Butler, M., Schulte, W. (eds.) FM 2011, *Lecture Notes in Computer Science* **6664**, 293–307.

Ciobanu, G. and Koutny, M. (2011) Timed Mobility in Process Algebra and Petri nets. *The Journal of Logic and Algebraic Programming* **80(7)**, 377–391.

Ciobanu, G. and Prisacariu, C. (2006) Timers for Distributed Systems. In Di Pierro, A. and Wiklicky, H. (Eds.) QAPL 2006,*Electronic Notes in Theoretic Computer Science* **164(3)**, 81–99.

Ciobanu, G. and Rotaru, A. (2013) A Probabilistic Logic for pTiMo. In Liu, Z., Woodcock, J. and Zhu, H. (Eds.) ICTAC 2013, *Lecture Notes in Computer Science* **8049**, 141–158.

Ciobanu, G. and Zheng, M. (2013) Automatic Analysis of TiMo Systems in PAT. In *Proc. 18th International Conference on Engineering of Complex Computer Systems (ICECCS 2013)*, IEEE Computer Society, 121–124.

Hennessy, M. (2007) *A Distributed π-calculus*. Cambridge University Press.

Henzinger, T.A., Nicollin, X., Sifakis, J. and Yovine,S. (1994) Symbolic Model Checking for Real-time Systems. *Information and Computation* **111**, 192–224.

Hessel, A., Larsen, K.G., Mikucionis, M. Nielsen, B. Pettersson, P. and Skou, A. (2008) Testing Real-Time Systems Using Uppaal. In Hierons, R.M., Bowen, J.P., Harman, M. (Eds.) FORTEST, *Lecture Notes in Computer Science* **4949**, 77–117.

Milner, R., Parrow, J. and Walker, D. (1992) A Calculus of Mobile Processes (i-ii). *Information and Computation* **100**, 1–77.

Moller, F. and Tofts, C. (1990) A Temporal Calculus of Communicating Systems. In Baeten, J.C.M., Klop, J.W. (Eds.) CONCUR 1990, *Lecture Notes in Computer Science* **458**, 401–415.

Reed, G.M. and Roscoe, A.W. (1988) A Timed Model for Communicating Sequential Processes. *Theoretical Computer Science* **58(1-3)**, 249–261.

Yi, W. , Pettersson, P. and Daniels, M. (1994) Automatic Verif cation of Real-time Communicating Systems by Constraint-solving. In *International Conference on Formal Description Techniques*, 223–238.