Workshop on Formal Methods for Aerospace (FMA)

Bujorianu, M.L. and Fisher, M.

2009

MIMS EPrint: **2010.51**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

Workshop on Formal Methods for Aerospace (FMA)

A workshop affiliated with **Formal Methods Week (FM2009)**

3rd November 2009, Eindhoven, Netherlands

Programme Chairs

Manuela Bujorianu
Michael Fisher

Steering Committee

Manuela Bujorianu	(Manchester, UK)
Michael Fisher	(Liverpool, UK)
Alessandro Giua	(Cagliari, IT)
Corina Pasareanu	(CMU and NASA, USA)

Programme Committee

Howard Barringer	(Manchester, UK)
Marius Bozga	(Verimag, FR)
Ricky Butler	(NASA, USA)
Ernst-Erich Doberkat	(Dortmund, DE)
Alessandro Giua	(Cagliari, IT)
Jianghai Hu	(Purdue, USA)
Rom Langerak	(Twente, NL)
John Lygeros	(Zurich, CH)
Savi Maharaj	(Stirling, UK)
Tiziana Margaria	(Potsdam, DE)
Cesar Munoz	(NASA, USA)
Flemming Nielson	(DTU, DK)
Dusko Pavlovic	(Oxford, UK)
Cristina Seceleanu	(Malardalen, SE)
Roberto Segala	(Verona, IT)
Ferucio Laurentiu Tiplea	(Iasi, RO)
Antonios Tsourdos	(Cranfield, UK)
Michael Whalen	(Minnesota, USA)
Virginie Wiels	(ONERA, FR)

Background

The coexistence of multiple disciplinary perspectives on the same class of critical applications (aerospace) and investigation (formal) methods leads naturally to the opportunity to define multidisciplinary approaches. Thus, work in this area will likely underline the importance of some research problems from aerospace to the formal methods community, and promote new formal techniques combining the principles of artificial intelligence and control engineering.

The source of new problems for formal methods comes from the great diversity of aerospace systems. These can be satellites, unmanned aerial vehicles (UAVs), terrestrial or other kinds of flying robots. These systems can be involved in complex activities such as space exploration, telecommunication support, fire detection, geo-mapping, weather prognoses, geo-rectification, search and rescue, naval traffic surveillance, tracking high value targets. From these applications, new research problems appear: autonomy, collective behaviour, information fusion, cognitive skills, coordination, flocking, etc. In addition, new concepts must be formalised: digital pheromones, swarms, system of systems of robots, sensing, physical actuation, and so on.

Aerospace systems are not only safety critical, but also mission critical and have very high performance requirements. For example, there is no safety issue regarding a planetary rover, but the system performance must justify the great cost of deploying it. Consequently, aerospace enriches traditional formal methods topics with new (or, at least, rarely investigated) research issues.

Formal methods could greatly benefit from integration with approaches from other disciplines, and many such opportunities are now appearing. A good example is the problem of coordination for platoons of UAVs or satellites, which have been successfully, tackled using various techniques from control engineering and numerical tools from dynamic programming. In addition, there exist an abundance of examples artificial intelligence techniques in aerospace (target tracking, rover planning, multi-agent technologies and so on). The implementation of these methods could benefit from formal development. From the cross-fertilization of related multidisciplinary approaches, we expect more robust, safe and mechanizable development and verification methods for aerospace systems.

Workshop

The main workshop objective is to promote a holistic view and interdisciplinary methods for design, verification and co-ordination of aerospace systems, by combining formal methods with techniques from control engineering and artificial intelligence. The very demanding safety, robustness and performance requirements of these systems require unprecedented integration of heterogeneous techniques and models. The aim of FMA is to bring together active researchers from all the above areas to discuss and present their work. Relevant topics include, but are not limited to (all with a focus on potential application in aerospace design or engineering):

- new modeling paradigms
- trajectory specification languages
- formal verification of safety and performance properties
- run-time monitoring
- combining formal and analytical techniques in modeling and verification
- autonomous systems
- heterogeneous and hybrid system models
- multi-agent coordination
- probabilistic and stochastic modeling and verification methods
- agent technologies

Format

The workshop format comprises three types of activities:

- Plenary lectures;
- Presentations of peer reviewed contributed papers;
- Contributed talks on ongoing research project or recent research trends that are not sufficiently disseminated across relevant academic communities.

Our plenary speakers are well known experts in the area of verification of space systems and area traffic management systems.

Klaus Havelund from the Jet Propulsion Laboratory at NASA will introduce the participants to the subtleties of run time verification of the log files for NASA's next 2011 Mars mission MSL (Mars Science Laboratory). The specific challenges include the design of

- a suitable specification language,
- verification algorithms of execution traces against formal specifications, and
- techniques of combining run time analysis with formal testing.

Henk Blom from the National Laboratory of Aerospace, Amsterdam, will introduce the audience into the world of verification for future air traffic management systems from a control engineering perspective. The techniques involved are of probabilistic nature and, in their most general form, are known as stochastic reachability analysis. Although the formal aspects of control engineering verification are less transparent, stochastic reachability analysis is the continuous counterpart of probabilistic model checking, which is a well established formal verification technique.

Three contributed papers have been selected for presentation. These describe novel approaches to three complementary topics in the area of formal development and verification with applications to aerospace systems. The contributed talks unite experts who very rarely attend a common scientific event. A major objective of the FMA workshop is the cross-fertilization of the multi-disciplinary approaches to the development and verification of aerospace systems. The abstracts of these talks indicate a possible common framework in the form of hybrid systems. The behaviour of these systems results from the interaction between discrete and continuous dynamics. Experts in the area of hybrid systems will interact with experts in software engineering in a quest for improved inter-disciplinary techniques for enhancing the quality of control software in avionics. In addition to formal verification, other topics specific to formal methods will be discussed, including observability, multi-agent systems, requirements analysis, synchronous languages, etc. Moreover, we hope that the formal methods experts will be challenged by unusual topics such as the stability of real-time control systems and stochastic abstractions for hybrid systems. Most importantly, experience reports on practical development of aerospace systems will be presented and these may provide insights, problems and benchmarks for future approaches.

We hope you enjoy the workshop . . . . . Manuela Bujorianu and Michael Fisher

Table of Contents

[Plenary] Runtime Verification of Log Files, a Trojan Horse for Formal Methods 1
Klaus Havelund, Howard Barringer, Alex Groce, David Rydeheard, Margaret Smith

[Contributed] Modelling and Verification of Multiple UAV Mission Using SMV 3
Gopinadh Sirigineedi, Antonios Tsourdos, Rafal Zbikowski, Brian A. White

[Plenary] Probabilistic Safety Verification of Future Air Traffic Management 15
Henk Blom

[Contributed] Implementing Multi-Periodic Critical Systems: from Design to Code Generation 16
Julien Forget, Frédéric Boniol, David Lesens, Claire Pagetti

[Contributed] Re-verification of a Lip Synchronization Protocol using Robust Reachability. 32
Piotr Kordy, Rom Langerak, Jan Willem Polderman

Agent Reasoning Beyond Logic - Abstractions of a 6DOF spacecraft 46
Nick Lincoln, Sandor Veres, Louise Dennis, Michael Fisher, Alexei Lisitsa

Observability and diagnosability of hybrid systems with application in Air Traffic Management 59
Maria Domenica Di Benedetto, Stefano Di Gennaro, Alessandro D'Innocenzo

Formalization and Validation of Safety-Critical Requirements 60
Alessandro Cimatti, Marco Roveri, Angelo Susi, Stefano Tonetta

Flexible Lyapunov Functions and Applications to Fast Mechatronic Systems 69
Mircea Lazar

Agent Based Approaches to Engineering Autonomous Space Software 73
Louise Dennis, Michael Fisher, Alexei Lisitsa, Sandor Veres, Nicholas Lincoln

Synoptic: a Domain Specific Modeling Language for embedded flight-software 76
Thierry Gautier

Developing Experimental Models for NASA Missions with ASSL 79
Emil Vassev, Mike Hinchey

Model Checking Nonlinear Hybrid Systems 86
Pieter Collins, Jan van Schuppen, Bert van Beek, Davide Bresolin, Ivan Zapreev, Sanja Zivanovic

Abstraction and Verification of Stochastic Hybrid Systems 87
Alessandro Abate

Runtime Verification of Log Files, a Trojan Horse for Formal Methods*

Howard Barringer[1], Alex Groce[3], Klaus Havelund[2], David Rydeheard[1], and Margaret Smith[2]

[1] School of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL, UK
{howard.barringer,david.rydeheard}@manchester.ac.uk

[2] Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
{klaus.havelund,margaret.h.smith}@jpl.nasa.gov

[3] School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, USA
alex.groce@eecs.oregonstate.edu

Runtime verification is the discipline of monitoring and analyzing program executions. A typical scenario consists of determining whether an execution trace satisfies a user-provided specification. Research challenges include development of expressive and convenient specification languages, development of decision procedures for fast analysis of traces against specifications, and minimization of impact on an instrumented running program being monitored. In this presentation, we motivate and show how a temporal rule-based runtime verification system has been applied to log file analysis in support of the software testing effort for NASA's next 2011 Mars mission MSL (Mars Science Laboratory).

RULER [2] is a general-purpose conditional rule-based system, which has a simple and easily implemented algorithm for effective runtime verification, and into which one can compile a wide range of temporal logics and other specification formalisms used for runtime verification. RULER has been designed with expressive power as well as specification convenience in mind. A RULER specification consists of a set of rules operating on a set of facts. A fact is of the form $F(v_1,...,v_n)$, where F is an identifier and $v_1,\ldots,v_n$ are values of various domains. For example, $FileSent(127)$ is a fact. Facts include observed events as well as internally generated *state*. A rule triggers when its condition (a predicate over the set of facts) is satisfied, and as a result the rule will add and/or remove facts from the set. Specifications can be parameterized with data, or even

* Part of the research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

with specifications, allowing for temporal logic combinators to be defined. The system has been developed in Java and can be directly applied to monitoring JAVA programs, using for example ASPECTJ [5] for code instrumentation.

The LOGSCOPE [4] system is a derivation from RULER, developed specifically for supporting testing of MSL flight software. It is implemented in PYTHON in order to integrate well with other test scripts written for MSL, also written in PYTHON. The LOGSCOPE specification language removes some of RULER's generality, resulting in the interesting subset of data parameterized state machines, and adding a simple user-friendly temporal logic. The temporal logic is mapped to the core parameterized state machines. A description of the process of introducing LOGSCOPE as part of the MSL testing effort is presented in [3]. The system has been used by test engineers to analyze log files generated by running the flight software. The temporal logic was instrumental in achieving test engineer acceptance of the technology. Detailed logging is already part of the MSL system design approach, and hence there is no added instrumentation overhead caused by this approach. While post-mortem log analysis prevents the autonomous reaction to problems possible with *online* runtime verification, it provides a powerful tool for test automation.

A combined presentation of the two systems is presented in [1]. We will conclude with a brief mention of the current effort to unify these two systems, providing more expressive temporal capability in rule specifications.

References

1. H. Barringer, K. Havelund, D. Rydeheard, and A. Groce. Rule systems for runtime verification: A short tutorial. In S. Bensalem and D. Peled, editors, *9th international workshop on Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 1–24, Grenoble, France, July 2009. Springer.
2. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *Journal of Logic and Computation*, 2009. Advance Access published on November 21, 2008. doi:10.1093/logcom/exn076.
3. A. Groce, K. Havelund, and M. Smith. From scripts to specifications, the evolution of a flight software testing effort. October 2009. Submitted for conference publication.
4. A. Groce, K. Havelund, M. Smith, and H. Barringer. Let's look at the logs: Low-impact runtime verification. *Computer Journal*, July 2009. Submitted for review.
5. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

Modelling and Verification of Multiple UAV Mission Using SMV

Gopinadh Sirigineedi[*], Antonios Tsourdos[†], Rafał Żbikowski[‡], and Brian A. White[§]

Department of Informatics and Sensors,
Cranfield University, Shrivenham,
Swindon SN6 8LA, United Kingdom.

Model checking has been used to verify the correctness of digital circuits, security protocols, communication protocols, as they can be modelled by means of finite state transition model. However, modelling the behaviour of hybrid systems like UAVs in a Kripke model is challenging. This work is aimed at capturing the behaviour of an UAV performing cooperative search mission into a Kripke model, so as to verify it against the temporal properties expressed in Computational Tree Logic (CTL). SMV model checker is used for the purpose of model checking.

1 Introduction

Increase in computational power, improvements in control techniques and other technological advances have led to increased focus on cooperative control of multiple agents in recent years. Cooperating multi-agent systems find application in large number of areas - mobile robots, micro satellite clusters, unmanned aerial vehicles (UAVs), automated highway systems and internet agents. Cooperating multiple-agents offer a large number of benefits such as: increase in the success rate of mission, large area coverage by improvements in latency and information gathering, increase in the computational power offered by distributed computing and graceful degradation in performance

Cooperative UAV control problems that have received recent attention include cooperative formation [17], cooperative task allocation, cooperative path planning, cooperative search and many others. Cooperative search problems have applications in a number of military and civilian applications, such as surveillance and reconnaissance operations, search and rescue; hazard monitoring, battle damage assessment, agricultural coverage tasks and security patrols [26].

Due to the mission critical nature of UAV systems, it is highly important to ensure the correctness of these systems and check if the system meets the design requirements. The failure of control software of the Arian-5 rocket and the Mars rover are hard remainders of what can happen when systems don't perform as per specifications. Verification is the process of verifying the correctness of the system and whether it satisfies the specifications. The common verification processes are *simulation, testing, deductive verification*, and *model checking* [14]. Simulation is performed on the abstract model of the system, where as testing is done on the actual system. Simulation and testing involve giving certain inputs and checking whether the outputs are as expected. These methods are cost effective way to find the errors. However, checking all of the possible interactions and possible faults is almost impossible

[*]PhD Student., DoIS, Cranfield University.
[†]Professor and Head of the Autonomous Systems Group., DoIS, Cranfield University.
[‡]Professor., DoIS, Cranfield University.
[§]Professor., DoIS, Cranfield University.

Submitted to:
Formal Methods for Aerospace 2009

using simulation and testing. They only ensure that the system works for the inputs they are tested for. Testing will demonstrate the presence of bugs, but will not demonstrate the absence of bugs. Even if the system passes all the testing, we can't claim that the system is completely free from errors, as no amount of testing is exhaustive enough.

The mission planning software of multiple UAV systems involves concurrency as it deals with multiple UAVs. It is also reactive, as it constantly interacts with the environment in which the UAVs operate. It is impossible to completely verify such a software system using traditional testing. In addition, concurrency bugs are one of the most difficult ones to test in a traditional way. Moreover, autonomous systems operate in harsh and unpredictable environments and it is difficult to predict before hand the kind of situations that may arise during the mission to carry out testing [7] [5] [20]. Hence, there is a need for formal verification methods, like deductive verification and model checking, which can clarify with high degree of certainty that the system meets its requirements. NASA has been working on developing formal verification techniques for their intelligent autonomous system involving multiple rovers or satellites [8] [18].

Deductive verification is proof-based. It refers to axioms and proof rules to prove the correctness of the system. System description is made in some formal language and leads to a set Γ of appropriate formulas in an appropriate logic. The set Γ constitutes a formal logical inference system for deduction. A system specification is an another formula φ of a chosen logic. The verification consists of finding a proof within the given formal logical system, which would demonstrate that the specification formula φ is inferred from the axioms and inference rules of the formal system Γ, *ie* $\Gamma \models \varphi$. The formal system Γ is assumed to be sound and complete. Deductive verification is well recognized in computer scientists and has significantly influenced the area of software development. However, deductive verification is time-consuming and can be performed only by experts in logic and mathematics.

Model checking, as the name suggests, is model-based. It is an automatic technique for verifying finite state systems. It involves developing a simplified model which captures the essential features of the systems. The specifications which are to be verified on the system are specified, usually in terms of logical statements. Then a model checker, a software tool, systematically examines all the system scenarios to check whether the system satisfies the specifications [4].

In [10] SPIN model checker has been used to verify the safety properties of multi-robot system expressed in Linear Temporal Logic (LTL). In [19] timed automata framework has been used to model the robots and Uppaal model checker has been used to verify the properties of multiple-robot system expressed in Computational Tree Logic (CTL). This paper presents formal modelling of multiple-UAV mission by means of Kripke model and verification of some of the mission properties expressed in CTL. Kripke model offers benefits of using graph theoretic approaches to analyze the system model. SMV model checker is used for verifying the properties, as it is one of the most popular model checkers that supports CTL. In our previous work [23], we reported Kripke model of the behaviour of a single UAV performing a search and verification of its properties expressed in CTL.

The rest of the paper is organized as follows: Section-2 gives an overview of model checking technique. The multiple UAV mission and single UAV behaviour performing the search are discussed in Section-3. Verification of the UAV behaviour using SMV model checker is presented in Section-4.

2 Overview of Model Checking

Model checking is a technique to verify finite state machine abstraction of the system. The system is represented by finite state model M and a temporal logic formula ϕ expressing some desired specification.

A model checker is used to check M against the specification ϕ. The model checkers outputs either true, if M satisfies ϕ *i.e* $M \models \phi$, or a counter example, if it does not. These different steps of model checking are discussed in detail below.

2.1 Model

The first step in model checking is to construct a *formal model* of the system. As model checking can be performed only on finite state systems, the system should be represented as a finite state transition diagram. We are primarily concerned with reactive systems like UAV systems and their behaviour over time. Reactive systems are systems which maintain constant interaction with the environment in which they operate. The family of reactive systems include many classes of programs whose correct and reliable construction is particularly challenging, including concurrent programs, embedded and process control programs, and operating systems. Typical examples of such systems are air traffic control systems, operating systems, and perpetual ongoing processes such as a nuclear reactors.

Reactive systems need to interact with the environment frequently and often do not terminate. Therefore, they can't be adequately modelled by input-output behaviour. Reactive systems can be modelled by capturing the features by means of *state*. A state is an instantaneous description of the system that captures the variables at a particular instant of time. The change from one state to the other as a result of some action determines the *transition* of the system. A *computation* is an infinite sequence of states where each state is obtained from the previous state by some transition.

Kripke structure or *Kripke model* is a type of state transition graph to capture this intuition about the behaviour of reactive systems. Kripke structure consists of a set of states, a set of transitions between the states, and a function that labels each state with a set of properties that are true in that state. Paths in Kripke structure model computations of the system. Although the model of the system is abstract and simple, it should be expressive enough to capture the aspects of temporal behaviour for reasoning about the system. Formal representation of Kripke structure is given below.

A Kripke model is represented by a triplet $M = (S, R, L)$ over a set of atomic propositions AP [14]. A concise representation of a Kripke model whose nodes are states is shown in Figure 1.

1. S is a finite set of states.
2. $R \subseteq S \times S$ is the transition relation.
3. $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

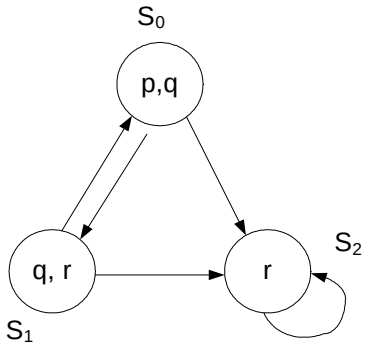


Figure 1: Kripke model representation of a finite state system

2.2 Specification

The properties of the system are usually specified in temporal logic. Temporal logic is a formalism for describing sequences of transitions between states in a reactive system. It is used for specifying and verifying the correctness of digital circuits and computer programs. In classical logic, such as predicate logic, the truth value of a statement is independent of time, whereas in temporal logic the truth value changes dynamically with time. As we are trying to capture the behaviour of multiple UAV group over time, temporal logic suits the purpose of specification language for specifying requirements of the system. There are two fundamental representation types of temporal logic: CTL and LTL. The distinction is how they handle the underlying computation tree.

CTL considers branching of time and allows future paths at any given point of time. The temporal operators quantify over the paths that are possible from a given state. The CTL operators are AX, EX, AG, EG, AU, EU, AF and EF. These operators are pairwise operators. The first of the pair is either A or E. A represents 'along all paths' and E represents 'there exists at least one path'. The second one of the pair is X, F, G, or U meaning 'next state', 'some future state', 'all future states(globally)' and 'until' respectively. CTL has the following syntax given in Backus Naur form:

$$\phi := \bot|\top|p|(\phi)|(\neg\phi)|(\phi\wedge\phi)|(\phi\vee\phi)|\phi\rightarrow\phi|AX\phi|EX\phi|A[\phi\cup\phi]|E[\phi\cup\phi]|AG\phi|EG\phi|AF\phi|EF\phi, \quad (1)$$

where p ranges over atomic formulas.

2.3 Verification

Many automatic model checkers are available, e.g. SMV [15], SPIN[9], KRONOS, HYTECH, NuSMV. SMV model checker has been developed by McMillan in the 90s. It uses SMV language for description of the system model. It accepts temporal specifications expressed either in CTL or LTL. SMV has been used for model checking digital circuits [11], security protocols [13], embedded systems [12] and web applications [16].

3 Mission planning for Multiple UAVs

In recent years there has been a growing interest in employing UAV teams to cooperatively search a given area. The operations of such groups include reconnaissance, surveillance, battle damage assessment, fire monitoring and chemical cloud detection. UAVs are suitable for these operations as they are too dangerous for human pilots. Some of the tasks such as monitoring forest fire propagation and mapping chemical cloud propagation can only be carried out by a group of UAVs and can't be carried out by a single UAV [21], [6], [25], [26].

Hierarchical approach is used extensively for cooperative control because of its simplicity and ease of design [21] [1] [24]. Each layer has different functionalities. This is done to simplify design and to deal with different aspects of the systems in separate layers. Partitioning may result in sub-optimal solution, but as each of these layers have different bandwidths dealing them separately can be justified. The decision making layer operates at very low rate, whereas the path planning layer operates at moderate rate and auto-pilot operates at high rate. The control architecture for multiple UAV mission is shown in Figure 2. The top layer performs as decision making layer taking inputs regarding the state of the UAVs from the middle layer. The middle layer is the path planning and guidance layer. The bottom layer consists of controllers for each UAV.

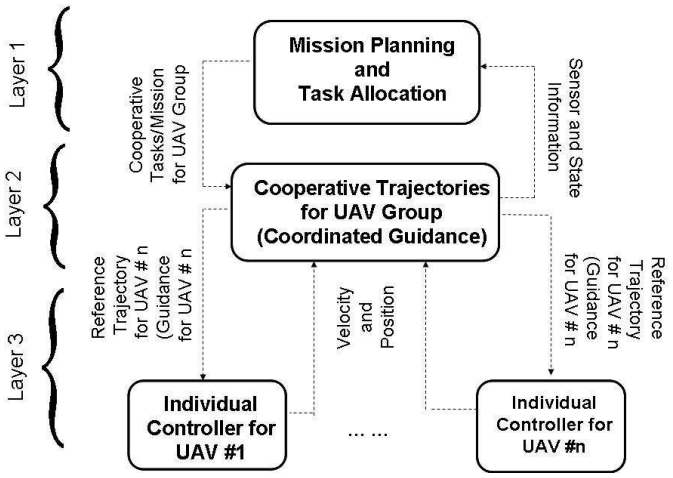


Figure 2: Architecture for multiple-UAV cooperative control

3.1 Cooperative UAV Search

This research is aimed at developing a verifiable multiple UAV mission for cooperatively searching a given area. The mission is to search a given bounded area for static targets and threats using a group of UAVs for the purpose of environmental monitoring. No or little information is available about the area being searched. The UAVs must cooperatively search the environment and mark the positions of targets and threats. Each UAV has a communication link to broadcast the findings and to receive updates from other UAVs of the group. Each UAV has sensors to monitor the environment for the presence of threats and targets. It is assumed that inter-UAV communications are instantaneous, noiseless and have unbounded communication range. It is assumed that each UAV has sufficient processing power for path planning and enough storage to store the global picture of the area being searched. The velocity of the UAVs is 20 m/s and the minimum turn radius is 25 m.

Each UAV stores on board a model of the environment in form of a “search map”. The positions of targets, positions of threats identified by the UAVs in the group and decisions of the other UAVs in the group are stored in this search map. This search map is constantly updated to reflect new information gathered and changes in the state of the UAVs. Sharing information is essential to ensure cooperation for decentralized control approach [2], [26]. As no centralized control is present, sharing of information among the UAVs helps in achieving cooperation. The environment being searched is divided into square cells. Based on the information in search map each UAV will identify an adjacent cell free from threats and other UAVs. The path planning layer generates a path starting from its present position, to the selected neighbouring cell with selected intial and destination headings.

The area to be searched is a square area of 2000X2000 metres. The area is discretized into square cells of 100x100 metres. Discretizing the area into cells helps in reducing the state space and also to visualize the state of location of the UAV. Each cell is identified by coordinates of the centre of cell. The UAVs move through the search area by selecting one of the neighbouring cells. The neighbouring five cells around the cell, in which UAV is flying, are marked as shown in Figure 3 for the purpose of identification. The cell with arrow corresponds to the cell in which the UAV is flying at the time of decision making and the direction of arrow indicates the current heading of the UAV. Cell marked with 1 corresponds to the cell just ahead in the direction of current heading. When cell1 is free, the UAV moves

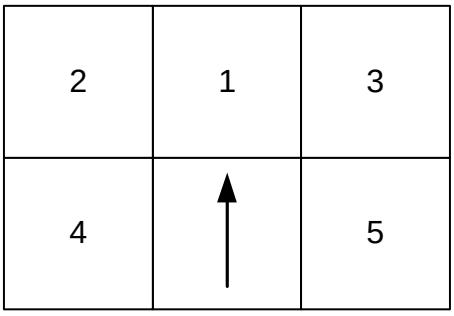


Figure 3: Neighbouring cells marked for decision making

into it. If cell1 contain a threat or it is already chosen by other UAV in the group, the UAV moves into cell3. The order of preference in selecting a neighbouring cell in decreasing order is - cell1, cell3, cell5, cell2 and cell4. When the UAV reaches the north-most cell it moves into cell5. When the UAV reaches the south-most cell it moves into cell4. The UAV flies in a path which connects the centre of present cell and the centre of the chosen neighbouring cell. The initial heading and destination headings of the path are either 90° or 270°. This is done to discretize the heading of the UAV to just two values in order to capture the UAV heading in finite state transition model. The heading is measured with respect to the east. Heading of 90° corresponds to the UAV flying north. The destination heading of the path is same as the initial heading if the UAV selects cell1 or cell2 or cell3. If not, the destination heading is opposite direction to the initial heading. Each UAV in the group repeats this behaviour of selecting a neighbouring free cell.

The decision making layer passes the information of the current and destination cells, current and destination headings to the path planning layer as shown in Figure 4. The path planning layer takes inputs from the decision making layer and generates a flyable path from the current cell to the destination cell with specified starting heading and destination heading. The flight dynamics of the UAV are not taken into account. It is assumed that the flight controller present on the UAV will take inputs from the path planning and guidance layer and follows the generated path.

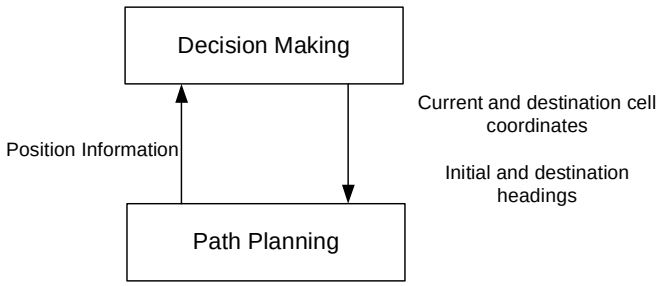


Figure 4: Information exchange between decision making layer and path planning layer

Dubins paths are used to generate path between the way points identified by the decision making layer. The decision making layer passes the inputs to the path planning layer which then produces a Dubins path for the initial and destination poses. Dubins path produces the shortest path between two points by concatenation of circular arcs and their connecting tangents [3] [22]. A straight line is the shortest path between two points. However, for UAVs with constraints on initial heading and final

heading and minimum turning radius, the shortest path is given by concatenation of circular arcs and straight line [3]. Four Dubins curve types *LRL*, *RSR*, *RSL*, *LSR* have been used. *L*, *S*, *R* denote turning left, straight line and turning right respectively. It is assumed that the UAVs fly with a constant altitude. Hence, 2D Dubins paths are used for path planning. 2D Dubins paths for different turning radii and different heading angles are shown in Figure 5.

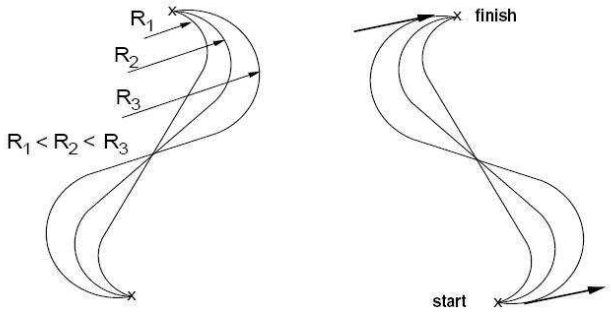


Figure 5: 2D Dubins paths for different turning radii and heading angles

The search strategy discussed above is implemented in `MATLAB`. The trajectories of two UAVs in the group for a flying time of 600 seconds is shown in Figure 6. The red dots indicate the cells with threats.

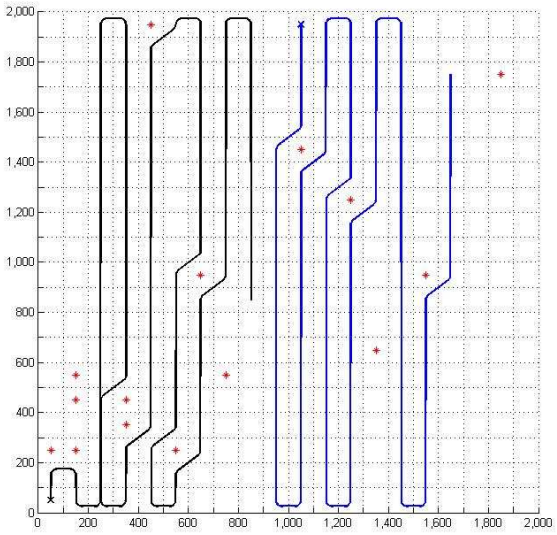


Figure 6: Simulation result of multiple UAV mission

4 Model Checking UAV Misssion

4.1 Kripke Model of UAV Mission

The behaviour of an UAV in the group performing the mission has been captured in Kripke model. CTL has been used to specify the properties because of its expressiveness and ease of modelling in Kripke model. As SMV model checker is a popular model checker for checking CTL properties is has been chosen for our work. SMV language is used for the description of the corresponding Kripke model.

The state of the UAV is captured by the current values of the programs variables. The state of the position of the UAV is captured by means of the coordinates of the centre of the cell. State variables `current_cell` and `destination_cell` capture the present and destination cell coordinates respectively. The initial values for `current_cell`, `initial_heading` are assigned using the keyword `init` as shown in Figure 7. The movement of the UAV from one cell to the next cell is modelled by assigning

```
init(current_cell) := [10 , 10];
init(initial_heading):=90;
```

Figure 7: SMV code showing assignment of initial values

one of the five neighbouring cells to the state variable `cell`. The state variables `initial_heading` and `destination_heading` capture the initial and destination headings of the UAV path connecting the present cell and the destination cell. The destination heading of the path becomes the initial heading for the next path and the destination cell becomes current cell, once the UAV moves to the destination cell. The SMV code modelling these transitions is shown in the Figure 8.

```
next(initial_heading) := destination_heading;
next(current_cell) := destination_cell;
```

Figure 8: SMV code showing transitions

The state variables `north_cell` and `south_cell` model the inputs from the navigation system regarding the present position of the UAV in the search area. The values for `north_cell` and `south_cell` are assigned deterministically as shown in the Figure 9. The state variables like `threat_in_cell1` model

```
north_cell := case
                current_cell[2]=1950:1;
                1:0;
              esac;
south_cell := case
                current_cell[2]=50:1;
                1:0;
              esac;
```

Figure 9: SMV code showing assignment

the environment in which the UAV is operating and model the presence of threats around the UAV. As these variables are purely environmental and the UAV has no control over the location of threats, they

are declared as non-deterministic. The state variables like `other_uav_selected_cell1` model the decisions made by other UAVs in the group. These variables are declared as non-deterministic, as the decisions of the other UAVs are not determined by the state of UAV. As the UAV's decisions are based on the information from the sensors which senses the presence of threats in the immediate neighbourhood, only five cells around the UAV are considered as shown in the Figure 3. For example the presence of threat in the cell11 is modelled by assigning value 1 to the boolean state variable `threat_in_cell1`. The movement of the other UAVs in the group are captured by the boolean state variables. For example the selection of the any UAV in the group to move into cell1 is modelled by assigning 1 to the state variable `other_uav_selected_cell1`.

The destination heading is either 90° or 270° based on the present heading and the destination cell chosen. The coordinates of the destination cell depend on the selection of the neighbouring cell and the coordinates of the current cell. A snippet of SMV code showing the assignment of destination cell coordinates to the state variable `destination_cell` is shown in the Figure 10.

```
destination_cell:=
         case
           cell=cell1  : case
              initial_heading =90 : [current_cell[1] , current_cell[2]+100];
              initial_heading =270: [current_cell[1] , current_cell[2]-100];
                         esac;
           cell=cell2  : ..
                         ..
                         ..
               ..
               ..
         esac;
```

Figure 10: SMV code showing assignment of destination cell coordinates

4.2 Specification and Verification of Properties

CTL is used to express the properties that the UAV mission is expected to satisfy. The property that "*when the UAV is flying with a heading of 90° and not in the north-most cell, then it flies straight when there is no threat ahead and no other UAV has chosen cell1*" is expressed by the formula:

$$AG(\texttt{initial\_heading} = 90 \wedge \neg\texttt{threat\_in\_cell1} \wedge \neg\texttt{other\_uav\_selected\_cell1} \wedge \neg\texttt{north\_cell} \rightarrow \texttt{cell} = \texttt{cell1})$$

Similarly, formula for the case when the UAV heading is 270° is expressed as:

$$AG(\texttt{initial\_heading} = 270 \wedge \neg\texttt{threat\_in\_cell1} \wedge \neg\texttt{other\_uav\_selected\_cell1} \wedge \neg\texttt{south\_cell} \rightarrow \texttt{cell} = \texttt{cell1})$$

The safety property that "*the UAV doesn't enter a cell when either a threat is present or other UAV in the group has already chosen to enter that cell*" is specified by the following formula for the case of

cell1:

$$AG(\texttt{threat\_in\_cell1} \vee \texttt{other\_uav\_selected\_cell1} \rightarrow \neg(\texttt{cell} = \texttt{cell1}))$$

The property that "*the UAV has a initial heading of either 90° or 270° in its path*" is expressed by the following formula

$$AG(\texttt{initial\_heading} = 90 \vee \texttt{initial\_heading} = 270)$$

The Kripke model is verified against the above properties using SMV model checker. Intel 2.2 GHz processor is used for verificaion and resources used are: user time of 0.046875 sec and system time of 0.015625 sec. SMV produced a output of true for all the above properties. The situation when the UAV is deadlocked and cannot decide where to move next is found by means of violation of the property that "*the UAV chooses one of the five neighbouring cells*". This property can be expressed in CTL as the following formula

$$AG(\neg(\texttt{cell} = \texttt{no\_free\_cell}))$$

The violation of the above property is simulated and the Figure 11 shows a case when the UAV is deadlocked and cannot move any further, as all the neighbouring cells contain threats.

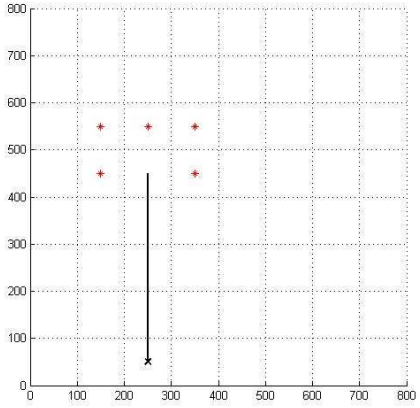


Figure 11: Simulation of a case showing deadlock

5 Conclusion and Future Work

The behaviour of an UAV performing multiple-UAV cooperative search is modelled by Kripke model and some of the properties of the mission are expressed in CTL. SMV model checker is used for verifying the correctness of the model against the temporal specifications. A deadlock has been found and the trace generated by SMV has been simulated. In future, concurrency issues, properties like area coverage, liveness, reachability and fairness have to be verified.

References

[1] Jovan D. Boskovic, Ravi Prakash & Raman K. Mehra (2002): *A multi-layer control architecture for unmanned aerial vehicles*. *Proceedings of the 2002 American Control Conference* , pp. 1825–1830.

[2] Wolfram Burgard, Mark Moors, Cyrill Stachniss & Frank E. Schneider (2005): *Coordinated multi-robot exploration*. *IEEE Transactions on Robotics* 21(3).

[3] L. E. Dubins (1957): *On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents*. *American Journal of Mathematics* 79(3), pp. 497–516.

[4] E.M.Clarke, E.A.Emerson & A.P.Sistla (1986): *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. *ACM Transactions on Programming Language and Systems* 8(2), pp. 244–263.

[5] Erann Gat (2004): *Autonomy Software Verification and Validation Might Not Be as Hard as it Seems*. *Proceedings of IEEE Aerospace Conference* , pp. 3123–3128.

[6] Paolo Gaudiano, Eric Bonabeau & Ben Shargel (2005): *Evolving behaviors for a swarm of unmanned air vehicles*. *Proceedings of the 2005 IEEE Swarm Intelligence Symposium* , pp. 317–324.

[7] Michael G.Hinchey, James L.Rash & Christopher A.Rouff (2002): *Verification and validation of autonomous systems*. *Proceedings of 26th Annual NASA Goddard Software Engineering Workshop* .

[8] Ari Jonsson Guillaume Brat (2005): *Challenges in verification and validation of autonmous systems for space exploration*. *Proceedings of IEEE International Joint Conference on Neural Networks* 5, pp. 2909–2914.

[9] G. J. Holzmann (1997): *The Model Checker Spin*. *IEEE Transactions on Software Engineering* 23(5), pp. 279–295.

[10] S. Jayarman, A. Tsourdos, R. Zbikowski & B. White (2006): *Kripke modelling approaches of a multiple robots systems with minimalist communication: a formal approach of choice*. *International Journal of System Sciences* 37(6), pp. 339–349.

[11] Daniela Kotmanova (2008): *Temporal logic in verification of digital circuits*. *Journal of Electrical Engineering* 59(1), pp. 14–21.

[12] Sandeep K.Shukla & Rajesh K.Gupta (2001): *A model checking approach to evaluating system level dynamic power management polocies for embedded systems*. *Proceedings of Sixth IEEE International High-Level Design Validation and Test Workshop* , pp. 53 – 57.

[13] Yuan Lu & Mike Jorda (2004): *Verifying a gigabit ethernet switch using SMV*. *Proceedings of the 41st Annual Conference on Design Automation* , pp. 230– 233.

[14] Edmund M.Clarke, Orna Grumberg & Doron A.Peled (1999): *Model Checking*. *The MIT Press* .

[15] K. L. McMillan (1999): *Cadence. Getting started with SMV*. *Cadence Berkeley Labs* .

[16] Huaikou Miao & Hongwei Zeng (2007): *Model checking-based verification of web applications*. *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems* , pp. 47–55.

[17] Chang-Su Park, Min-Jea Tahk & Hyochoong Bang (2003): *Multiple aerial vehicles formation using swarm intelligence*. *Proceedings of AIAA Guidance, Navigation, and Control Conference and Exhibit* .

[18] Charles Pecheur (2000): *Verification and validation of autonomy software at NASA*. *NASA/TM 2000-209602* .

[19] M. M. Quottrup, T. Bak & R. Izadi-Zamanabadi (2004): *Multi-robot planning: a timed automata approach*. *Proceedings of the 2004 IEEE International Conference on Robotics and Automation* .

[20] Christopher Rouff, Mike Hinchey, Walt Truszkowski & James Rash (2005): *Verifying large number of cooperating adaptive agents*. *Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS 2005)* 1, pp. 391–397.

[21] R.W.Beard, T.W.McLain & D.B.Nelson (2007): *Decentralized cooperative aerial surveillance using fixed-wing miniature UAVs*. *Proceedings of the IEEE* 94(7), pp. 1306–1324.

[22] A. M. Shkel & V. Lumelsky (2001): *Classification of the dubins set*. *Robotics and Autonomous Systems* 34(4), pp. 179–274.

[23] Gopinadh Sirigineedi, Antonios Tsourdos, Brian A. White & Rafał Żbikowski (2009): *Towards verifiable approach to mission planning for multiple UAVs*. *Proceedings of AIAA Infotech@Aerospace Conference and AIAA Unmanned .. Unlimited Conference* .

[24] Randal W.Beard, Tinmothy W.McLain, Michael A.Goorich & Erik P.Anderson (2002): *Coordinated target assignment and intercept for unmanned air vehicles*. *IEEE Transactions on Robotics and Automation* 18(6), pp. 911–922.

[25] Y.Jin, A.A.Minai & M.Polycarpou (2003): *Cooperative real-time search and task allocation in UAV teams*. *Proceedings of IEEE Conference on Decision and Control* , pp. 7–12.

[26] Y.Yang, M.Polycarpou & A.A.Minai (2007): *Multi-UAV cooperative search using an opportunistic learning method*. *Transactions of the ASME* 129, pp. 716–728.

Probabilistic safety verification of future air traffic management

Henk A.P. Blom, PhD
National Aerospace Laboratory NLR
Amsterdam, The Netherlands
blom@nlr.nl

Abstract
In spite of advances in formal and probabilistic verification approaches, fault and event trees still are the dominant techniques used for safety risk analysis in aviation. However, the combination of concurrent, dynamic and random effects that appear in air traffic cannot properly be captured by these classical techniques. In this lecture it will be explained how safety risk modeling and analysis can be formulated as a problem of estimation rare event probability of a large scale stochastic hybrid system. Subsequently it is explained how rare event estimation theory for diffusions can be extended to a large scale stochastic hybrid system (SHS) where a very huge number of rare discrete modes may contribute significantly to the rare event estimation. Essentially, the approach taken is to develop a compositional model of the air traffic operation considered in the form of a large scale SHS, subsequently to introduce a suitable aggregation of the discrete modes of this large scale SHS, and then to develop importance sampling and Rao-Blackwellization relative to these aggregations. The practical use of this approach is demonstrated for the estimation of mid-air collision probability for an advanced air traffic application.

Further reading:

1. H.A.P. Blom, B. Klein Obbink, G.J. Bakker, Simulated safety risk of an uncoordinated airborne self separation concept of operation, ATC-Quarterly, March 2009.
2. H.A.P. Blom, G.J. Bakker, J. Krystul, Rare event estimation for a large-scale stochastic hybrid system with air traffic application, Eds: G. Rubino and B. Tuffin, Rare event simulation using Monte Carlo methods, J.Wiley, pp. 193-214, 2009.
3. M.H.C. Everdij and H.A.P. Blom, Enhancing hybrid state Petri nets with the analysis power of stochastic hybrid processes, Proceedings 9th International Workshop on Discrete Event Systems (WODES), Göteborg, Sweden, May 2008, pp. 400-405.
4. H.A.P. Blom, J. Krystul, G.J. Bakker, M.B. Klompstra, B. Klein Obbink, Free flight collision risk estimation by sequential Monte Carlo simulation, Eds: C.G. Cassandras and J. Lygeros, Stochastic hybrid systems, Taylor & Francis/CRC Press, 2007, chapter 10, pp. 249-281.
5. J. Krystul, H.A.P. Blom, A. Bagchi, Stochastic differential equations on hybrid state spaces, Eds: C.G. Cassandras and J. Lygeros, Stochastic hybrid systems, Taylor&Francis/CRC Press, 2007, chapter 2, pp. 15-45.
6. M.H.C. Everdij, M.B. Klompstra, H.A.P. Blom, B. Klein Obbink, Compositional specification of a multi-agent system by stochastically and dynamically coloured Petri nets, Eds: H.A.P. Blom, J. Lygeros (eds.), Stochastic hybrid systems: Theory and safety critical applications, Springer, 2006, pp. 325-350.
7. M.H.C. Everdij, H.A.P. Blom, Hybrid Petri nets with diffusion that have into-mappings with generalized stochastic hybrid processes, Eds: H.A.P. Blom, J. Lygeros (eds.), Stochastic hybrid systems: Theory and safety critical applications, Springer, 2006, pp. 31-63.
8. M.H.C. Everdij and H.A.P. Blom, Piecewise deterministic Markov processes represented by dynamically coloured Petri nets, Stochastics, Vol. 77 (2005), pp. 1-29.

Implementing Multi-Periodic Critical Systems: from Design to Code Generation*

Julien Forget[1], Frédéric Boniol[1], David Lesens[2], and Claire Pagetti[2]

[1] ONERA, Toulouse, France, Email: firstname.lastname@onera.fr
[2] EADS Astrium Space Transportation, Les Mureaux, France

Abstract. This article presents a complete scheme for the development of Critical Embedded Systems with Multiple Real-Time Constraints. The system is programmed with a language that extends the synchronous approach with high-level real-time primitives. It enables to assemble in a modular and hierarchical manner several locally mono-periodic synchronous systems into a globally multi-periodic synchronous system. It also allows to specify flow latency constraints. A program is translated into a set of real-time tasks. The generated code (C code) can be executed on a simple real-time platform with a dynamic-priority scheduler (EDF). The compilation process (each algorithm of the process, not the compiler itself) is formally proved correct, meaning that the generated code respects the real-time semantics of the original program (respect of periods, deadlines, release dates and precedences) as well as its functional semantics (respect of variable consumption).

1 Introduction

Embedded systems have successfully been implemented with synchronous languages in the past. In particular, data-flow synchronous languages (Lustre/ Scade [6], Signal [1]) are well adapted for describing precisely the data flow between the communicating processes of the system. In [5] we proposed an extension of synchronous languages to design multi-periodic systems efficiently, by assembling several synchronous nodes into a multi-periodic synchronous program. Such an approach allows to describe the real-time aspects and the functional aspects of a system in the same framework. The purpose of the paper is to give an overview of the language capabilities and to describe the compilation chain through the programming of a case study. The generated code is targeted for a simple real-time platform with the *earliest-deadline-first* (EDF) scheduling policy [9]. We present the whole compilation chain but we do not get into the details of the proofs, which can be found in [4]. We focus more particularly on the generated code, which gives a concrete illustration of the compilation and summarizes our contribution.

* This work was funded by EADS Astrium Space Transportation

1.1 Motivation

The development of an industrial critical system may involve several teams, which separately define the different functions of the system. The functions are then assembled by the integrator, who describes the communications between the functions. Currently, this integration process lacks a formal language to ease the design process and to ensure the correctness of the global system.

We consider the simplified Flight Control System of Fig. 1 as a case study. This system controls the attitude, the trajectory and the speed of an airplane in auto-pilot mode. It consists of three communicating sub-systems. Each sub-system consists of several operations (represented by boxes in the figure) and executes repeatedly at a periodic rate. The fastest sub-system executes at 10ms, it acquires the state of the system (angles, position, acceleration) and computes the feedback law of the system. The intermediate sub-system is the piloting loop, it executes at 40ms and manages the flight control surfaces of the airplane. The slowest sub-system is the navigation loop, it executes at 120ms and determines the acceleration to apply. The required position of the airplane is acquired at the slow rate.

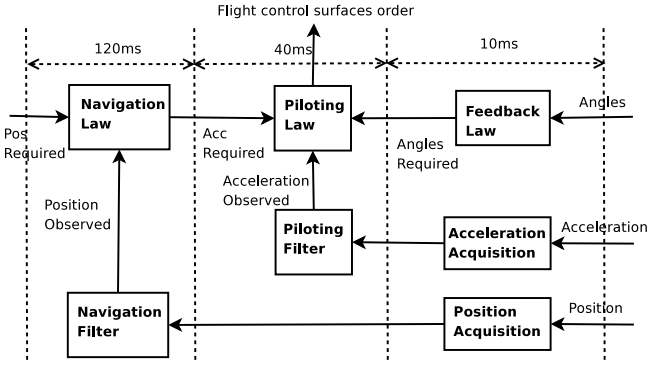


Fig. 1. Flight control system

The three sub-systems are first defined separately by different teams. The integrator then assembles them in the global system and specifies how they communicate. The language focuses on this assembly level.

1.2 Contribution

The main novelties of our approach are: first the integrator can develop the complete system in a unified formal framework (a high-level formal language) and second the language along with its compiler covers the development of the system from its design to its implementation, through automated code generation.

This relies on two different research domains. On the one hand, scheduling theory focuses mainly on satisfying system real-time constraints but usually disregards system functional behaviour. This ensures the correctness of the

temporal properties of the system, but makes it hard to verify the functional correctness of the system. In the case of multi-periodic systems, this often leads to non-deterministic process communications. On the other hand, synchronous languages focus on the correctness of the functional behaviour of the system and ensure that it is deterministic. However, classic synchronous languages abstract from real-time (except some recent extensions discussed in Sect. 7), which makes them ill-adapted to the implementation of multi-periodic systems.

Our work combines scheduling theory and synchronous languages to ensure both the functional and the temporal correctness of multi-periodic systems. This is particularly suitable for the implementation of critical systems. The integrator programs the system with the language introduced in [5], which extends synchronous languages with high-level real-time primitives. The compiler then generates the set of real-time tasks corresponding to this program. Tasks are then translated into C threads, each one containing the functional code of a task completed with a deterministic data-exchange protocol that does not require synchronization primitives (such as semaphores). The threads are scheduled concurrently with the EDF policy. They can be preempted by the scheduler during their execution but preemptions do not jeopardize the functional correctness of the system. The use of an EDF scheduler departs from the classic compilation scheme of synchronous languages, which translates a program into a "single-loop" sequential code [7]. The single-loop scheme relies on a static-priority non-preemptive scheduling policy, which makes it ill-adapted for implementing multi-periodic processes. A dynamic-priority preemptive policy like EDF allows to achieve better processor utilization (to execute more time-consuming processes). The complete compilation process has been implemented in OCaml and is about 3000 code lines long. It generates C code with calls to the real-time primitives defined in the real-time extensions of POSIX [13].

1.3 Paper Outline

Sect. 2 gives an overview of the language for specifying multi-periodic systems and shows how the case study can be programmed. We then detail the compilation process. The correctness of the system is first verified by a series of static analyses (Sect. 3). We then translate the program into a set of real-time tasks (Sect. 4). The preservation of the synchronous semantics during inter-task communications is ensured by a buffering protocol described in Sect. 5. We can then translate the tasks into C code for a simple real-time platform (Sect. 6). A comparison with related works is given in Sect. 7.

2 A Synchronous Real-Time Language

2.1 Informal Presentation

We present the language through the programming of the Flight Control System of Fig. 1. The different operations of the Flight Control System are first declared as imported nodes, named after the initials of the operations (for instance, `PA` stands for "Position Acquisition"):

```
imported node PA(i: int) returns (o: int) wcet 1;
imported node AA(i: int) returns (o: int) wcet 1;
...
```

The declaration of an imported node specifies the inputs and outputs of the node with their types and the worst case execution time (wcet) of the node. For instance, the node `PA` has one input `i` of type `int`, one output `o` of type `int` and its wcet is 1.

For each sub-system, we define an intermediate node that groups the operations of the sub-system. Node definitions are modular and hierarchical. There are several ways to decompose the Flight Control System into nodes and it is also possible to program the whole system as a single node. However, the different decompositions produce the same behaviour as intermediate nodes are flattened during the compilation (see Sect. 4.1). We choose to group operations by sub-systems (and by rates) for better readability. In the following, the suffix `_o` stands for "observed", `_r` for "required" and `_i` for "intermediate". The node for the acquisition loop is defined as follows:

```
node acquisition(angle,pos,acc)returns(pos_i,acc_i,angle_r)
let
  pos_i = PA(pos);
  acc_i = AA(acc);
  angle_r = FL(angle);
tel
```

This node has three inputs and three outputs, the types of which are left unspecified and will be inferred by the type-checker of the language (see Sect. 3). The body of the node (the `let ... tel` block) is a set of equations that define how the outputs of the node are computed from its inputs. All the variables and expressions of a program are flows. For example, the constant value 0 stands for an infinite constant sequence. Nodes are applied point-wisely to their arguments. So, for instance, at each repetition of node `acquisition`, the output `pos_i` is obtained by applying node `PA` to input `pos`.

Similarly, we define a node for the piloting loop and for the navigation loop:

```
node piloting (angle_r, acc_i, acc_r) returns (order)
var acc_o;
let
  acc_o = PF(acc_i);
  order = PL (angle_r, acc_o, acc_r);
tel
node navigation (pos_i, pos_r) returns (acc_r)
var pos_o;
let
  pos_o = NF(pos_i);
  acc_r = NL(pos_o,pos_r);
tel
```

So far, each node could be defined with the existing LUSTRE language as each sub-system is mono-periodic. For the main node `FCS` however, we use new primitives to handle *rate transitions* (when operations of different rates communicate) and to specify the real-time constraints of the different operations:

```
node FCS (pos_r: rate (120, 0); angle, pos, acc) returns (order: due 15)
var acc_i, acc_r, angle_r, pos_i;
let
  acc_r = navigation(pos_i/^12,pos_r);
```

```
  order = piloting(angle_r/^4, acc_i/^4, (0 fby acc_r)*^3);
  (pos_i, acc_i, angle_r) = acquisition(angle,pos,acc);
tel
```

When a faster node consumes a flow produced by a slower node, we under-sample the flow using operator $/\hat{}$. $e/\hat{}k$ only keeps the first value out of each k successive values of e. For instance flow `acc_i` is under-sampled by factor 4 as its consumer (`piloting`) is 4 times slower than its producer (`acquisition`).

For communications from slow to fast operations, we first delay the flow with operator `fby`. The operator `fby` inserts a unitary delay: expression *cst* `fby` e produces the value *cst* at its first iteration and then the previous values of e (ie e delayed by the period of e). We then over-sample the delayed flow with operator $*\hat{}$. $e*\hat{}k$ over-samples e by a factor k. Each value of e is duplicated k times in the result. For instance the flow `acc_r` is delayed and then over-sampled by a factor 3 as its consumer (`piloting`) is 3 times faster than its producer (`navigation`). We use a delay before over-sampling the flow to avoid reducing the deadline for the production of the flow. In the case of flow `acc_r` for instance, without a delay the deadline for `NL` would be lower than 40.

For now, we have only described the ratio between the execution rates of the nodes. The declaration of the node inputs simply specifies that `pos_r` has clock $(120, 0)$ (ie that it has period 120 and phase 0) and all the different rates of the system are deduced from this information by the clock calculus (see Sect. 3). The declaration of output `order`, imposes a deadline constraint (`due 15`), which requires it to be produced less than 15ms after the beginning of its period, to respect some external environment constraint. Its period is left unspecified (its inferred value is 40ms). The behaviour of the new operators is illustrated in Fig. 2, in which we give the value of each expression at each repetition of the system.

date	0	10	20	30	40	...
`angle_r`	an_0	an_1	an_2	an_3	an_4	...
`angle_r/^4`	an_0				an_4	...
date	0	40	80	120	160	...
`acc_r`	ac_0			ac_1		...
`0 fby acc_r`	0			ac_0		...
`(0 fby acc_r)*^3`	0	0	0	ac_0	ac_0	...

Fig. 2. Behaviour of real-time operators

2.2 Formal Definition: Strictly Periodic Clocks

In the synchronous approach, the computations performed by the system are split into a succession of *instants*, where each instant corresponds to one repetition of the system. The synchronous assumption requires that for each instant, computations end before the end of the instant. Computations can be activated or deactivated at different instants using *clocks*. Clocks define the temporal behaviour of the program on the logical time scale of instants.

To define formally the real-time operators presented in the previous section, we introduce a new class of clocks called *strictly periodic clocks*. Given a set of *values* $\mathcal{V}$, a *flow* is a sequence of pairs $(v_i, t_i)_{i\in\mathbb{N}}$ where v_i is a value in $\mathcal{V}$ and t_i is a tag in $\mathbb{N}$, such that for all i, $t_i < t_{i+1}$. The clock of a flow is its projection on $\mathbb{N}$. A tag represents an amount of time elapsed since the beginning of the execution of the program. Each value of a flow must be computed before its next activation: v_i must be produced during the time interval $[t_i, t_{i+1}[$. After precedence encoding, the deadline may actually be less than t_{i+1}, this will be detailed in Sect. 4.3.

Definition 1. *(*Strictly periodic clock*). A clock $h = (t_i)_{i\in\mathbb{N}}$, $t_i \in \mathbb{N}$, is* strictly periodic *if and only if:* $\exists n \in \mathbb{N}^*,\ \forall i \in \mathbb{N},\ t_{i+1} - t_i = n$.

n is the period *of h, denoted $\pi(h)$ and t_0 is the* phase *of h, denoted $\varphi(h)$.*

Definition 2. *The term $(n, p) \in \mathbb{N}^* \times \mathbb{Q}^+$ denotes the strictly periodic clock α such that $\pi(\alpha) = n$ and $\varphi(\alpha) = \pi(\alpha) * p$*

A strictly periodic clock defines the real-time rate of a flow and is uniquely characterized by its phase and by its period. Strictly periodic clocks relate logical time (instants) to real-time. Locally, each flow has its own notion of instant (it must end before its next activation), and globally we can compare flows that do not share the same notion of instants by relating instants to real-time. We introduce periodic clock transformations to formalize such rate transitions:

Definition 3. *Let α be a strictly periodic clock, operations $/_.$, $*_.$ and $\rightarrow_.$ are periodic clock transformations, that produce new strictly periodic clocks satisfying the following properties:*

- $\pi(\alpha /_. k) = k * \pi(\alpha),\ \varphi(\alpha /_. k) = \varphi(\alpha), k \in \mathbb{N}^*$
- $\pi(\alpha *_. k) = \pi(\alpha)/k,\ \varphi(\alpha *_. k) = \varphi(\alpha), k \in \mathbb{N}^*$
- $\pi(\alpha \rightarrow_. q) = \pi(\alpha),\ \varphi(\alpha \rightarrow_. q) = \varphi(\alpha) + q * \pi(\alpha), q \in \mathbb{Q}$

Rate transition operators apply periodic clock transformations to flows. If e has clock α, then $e/\hat{}\, k$ has clock $\alpha /_. k$, $e *\hat{}\, k$ has clock $\alpha *_. k$ and $e \sim> q$ has clock $\alpha \rightarrow_. q$.

2.3 Syntax

The syntax of the language is close to Lustre. It is extended with real-time primitives based on strictly periodic clocks. The grammar of the language is given in Fig. 3. A program consists of a list of node declarations (*nd*). Nodes can either be defined in the program (`node`) or implemented outside (`imported node`), for instance by a C function. Node durations must be provided for each imported node, more precisely an upper bound on worst case execution times (*wcet*). The external code provided for imported nodes can itself be generated by a standard synchronous language compiler (like Lustre), in case developers want to program the whole system using synchronous languages. The clock of input/output

parameters (in/out) of a node can be declared strictly periodic ($x : rate(n, p)$, x then has clock (n, p)) or unspecified (x). A deadline constraint can be imposed on outputs ($x : rate(n, p)$ `due` n', the deadline is n'). The body of a node consists of an optional list of local variables (var) and a list of equations (eq). Each equation defines the value of one or several variables using an expression on flows ($var = e$). Expressions may be immediate constants (cst), variables (x), pairs ((e, e)), initialised delays (cst `fby` e), applications ($N(e)$) or expressions using strictly periodic clocks ($epck$). Values k, n, n', q must be statically evaluable. Value q must be an element of $\mathbb{Q}^+$.

$$
\begin{array}{lll}
cst & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid ... \\
var & ::= & x \mid var, var \\
e & ::= & cst \mid x \mid (e, e) \mid cst \texttt{ fby } e \mid N(e) \mid epck \\
epck & ::= & e/\hat{}\,k \mid e *\hat{}\, k \mid e \sim> q \\
eq & ::= & var = e \mid eq; eq \\
in & ::= & x : rate(n, p) \mid x \mid in; in \\
out & ::= & x : rate(n, p) \mid x : rate(n, p) \texttt{ due } n' \mid x \mid out; out \\
nd & ::= & \texttt{node } N(in) \texttt{ returns } (out)[\texttt{ var } var;] \texttt{ let } eq \texttt{ tel} \\
 & & \mid \texttt{imported node } N(in) \texttt{ returns } (out) \texttt{ wcet } n;
\end{array}
$$

Fig. 3. Language grammar

3 Static Analyses

Synchronous languages are targeted for critical systems. Therefore, the compilation process puts strong emphasis on the verification of the correctness of the programs to compile. This consists of a series of static analyses of the program, which are performed before code generation.

The first analysis performed by the compiler is the type-checking. The language is a strongly typed language, in the sense that the execution of a program cannot produce a run-time type error. Each flow has a single, well-defined type and only flows of the same type can be combined. The type-checking of the language is fairly standard [12]. For the example of the Flight Control System, the type-checker produces the flowing type for node `FCS`: `(int*int*int*int)->int`. This means that the node takes four integer inputs and produces one integer output. This type is inferred from the types of the imported nodes.

The causality check verifies that the program does not contain cyclic definitions: a variable cannot instantaneously depend on itself (i.e. not without a `fby` in the dependencies). For instance, the equation `x=x+1;` is incorrect, it is similar to a deadlock since we need to evaluate `x+1` to evaluate `x` and we need to evaluate `x` to evaluate `x+1`.

The clock calculus (defined in [5]) verifies that a program only combines flows that have the same clock. When two flows have the same clock, they are *synchronous* as they are always present at the same instants. Combining non-synchronous flows leads to non-deterministic programs as we access to undefined values. For instance we can only compute the sum of two synchronous flow, because the value of the sum is ill-defined when only one of

the two flows is absent (when the two flows are absent the sum flow is simply absent, which is well-defined). The clock calculus ensures that a synchronous program will never access to undefined values. For the example of the Flight Control System, the clock-calculus produces the following clock for node `FCS`: `((120,0)*(10,0)*(10,0)*(10,0))->(40,0)`. This means that inputs `angle`, `acc`, `position` have period 10, while input `position_r` has period 120. The output `order` has period 40 (though its deadline is 10). These static analyses ensure that a program accepted by the compiler has a deterministic behaviour.

4 Translation into Real-Time Tasks

This section details how the compilation process translates the input program into a set of real-time tasks. We first extract a task graph from the program, where tasks are related by precedence constraints. We then encode task precedences in task real-time attributes to obtain a set of independent tasks.

4.1 Task Graph Extraction

Tasks A synchronous program consists of a hierarchy of nodes, the leaves of which are predefined or imported nodes. The task graph generation process first inlines intermediate nodes appearing in the main node recursively, replacing each intermediate node call by its set of equations. For instance, the program of the Flight Control System of Sect. 2.1 is translated into a single node (the main node `FCS`) containing one node call to each imported node, `PA`, `AA`, `FL`, `PF`, `PL`, `NF`, `NL`, one node call to operator $*\hat{}$, one node call to operator `fby` and three node calls to operator $/\hat{}$.

This "flattened" main node is then translated into a task graph. Each imported node call is translated into a task. Each variable of the node and predefined operator call is also translated into a vertex but will later be reduced to simplify the graph (see Sect. 4.1). The clustering of several nodes into the same task to reduce the number of generated tasks is beyond the scope of this paper. We could probably reuse existing strategies, for instance those suggested in [3].

Task Precedences In order to respect the synchronous semantics, for each data-dependency there must be a precedence from the task producing the data to the task consuming it. Task precedences are deduced from data dependencies between expressions of the program. Similarly to [7], we say that an expression e' precedes an expression e when e syntactically depends on e'. This occurs either when e' appears in e or when x appears in e and we have an equation $x = e$. Let $g = (V, E)$ denote a task graph, where V is the set of tasks of the graph and E is the set of task precedences of the graph (a subset of $V \times V$). For instance, the flattened Flight Control program contains the two equations `pos_o=NF(pos_i/^12); pos_i=PA(pos);`. These equations produce the following graph: ({`NF,PA,/^12,pos,pos_i,pos_o`}, {`pos`→ `PA`, `PA`→ `pos_i`, `pos_i`→ `/^12`, `/^12`→ `NF`, `NF` → `pos_o`}).

Task Graph Reduction We then simplify the intermediate graph structure. First, each input of the main node is translated into a (sensor) task and each output of the main node is translated into a (actuator) task. Second, we remove variables from the graph, replacing recursively each pair of precedence $N \rightarrow x \rightarrow M$, where x is a local variable, by a single precedence $N \rightarrow M$.

We finally translate predefined nodes into *precedence annotations*. A precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$ represents an *extended precedence*, where *ops* is a list of precedence annotations *op*, with $op \in \{*\hat{}\, k, /\hat{}\, k, \sim\!> q, \texttt{fby}\}$. *op.ops* denotes the list whose head is *op* and whose tail is *ops* and ϵ denotes the empty list. For instance, the precedences `PA`$\rightarrow$ `pos_i`, `pos_i`$\rightarrow$ `/^12`, `/^12`$\rightarrow$ `NF` are simplified into a single extended precedence `PA`$\overset{/\hat{}\,12}{\rightarrow}$`NF`. When the rewriting terminates, every task of the graph corresponds to either an imported node or a sensor or an actuator. The reduced task graph of the Flight Control System is given in Fig. 4.

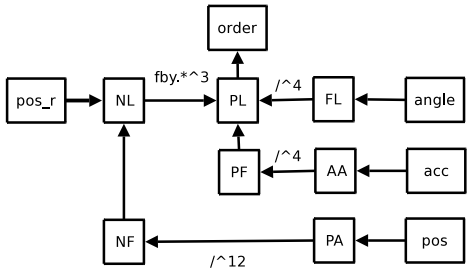


Fig. 4. Reduced task graph for the Flight Control System program

4.2 Real-Time Characteristics Extraction

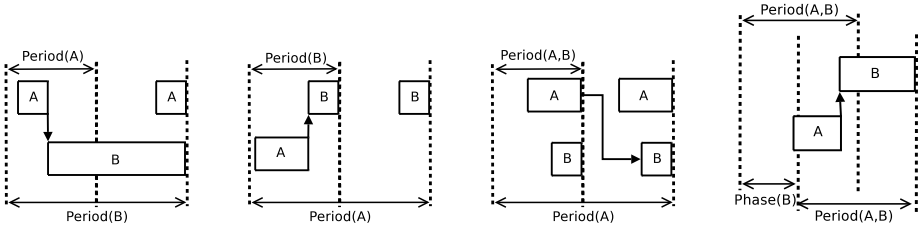
Each task τ_i of the graph is characterized by its real-time attributes (T_i, C_i, r_i, d_i). τ_i is instantiated periodically with period T_i. It cannot start its execution before all its predecessors, defined by the precedence constraints, complete their execution. C_i is the (worst case) execution time of the task. r_i is the release date of the first instance of the task. The subsequent release dates are $r_i + T_i$, $r_i + 2T_i$, etc. d_i is the relative deadline of the task. The absolute deadline $D_i[j]$ of the instance j of a task τ_i is the release date $R_i[j]$ of this instance plus the relative deadline of the task: $D_i[j] = R_i[j] + d_i$. Task real-time characteristics are extracted as follows:

- *Periods*: The period of a task is obtained from its clock ck_i. We have $T_i = \pi(ck_i)$.
- *Deadlines*: By default, the deadline of a task is its period ($d_i = T_i$). Deadline constraints can also be specified on the production of a node output (`o: due n`).
- *Release Dates*: The initial release date of a task is the phase of its clock: $r_i = \varphi(ck_i)$.
- *Execution Times*: The execution time C_i of a task is directly specified by the `wcet` of the imported node declaration. For simplification, we consider that the run-time overhead due to task preemptions is negligible.

4.3 Precedence Encoding

Simple Precedences Encoding [2] showed that a set of dependent tasks (task related by precedence constraints) can be reduced to a set of independent ones (without precedences) obtaining an equivalent problem under the EDF policy, by adjusting task release dates and deadlines such that precedences are encoded in the adjusted real-time characteristics. The adjusted absolute deadline D_i^* of a task is: $D_i^* = \min_{\tau_j \in succ(\tau_i)}(D_i, min(D_j^* - C_j))$. If we want to perform a schedulability test, the adjusted release date of a task is: $R_i^* = \max_{\tau_j \in pred(\tau_i)}(R_i, max(R_j^* + C_j))$. If we only want to schedule the program correctly, the adjusted release date of a task $R_i^{*'}$ is: $R_i^{*'} = \max_{\tau_j \in pred(\tau_i)}(R_i, max(R_j^{*'}))$. For simplification, we only consider the second encoding in the following.

Extended Precedences Encoding Fig. 5, shows that we can unfold extended precedences between tasks into simple precedences between task instances. The precedence encoding technique can then be applied to the "unfolded" graph.



(a) $A \overset{/\hat{}2}{\rightarrow} B$ (b) $A \overset{*\hat{}2}{\rightarrow} B$ (c) $A \overset{\text{fby}}{\rightarrow} B$ (d) $A \overset{\sim>1/2}{\rightarrow} B$

Fig. 5. Encoding extended precedences

More formally, let $\tau_i[n] \rightarrow \tau_j[n']$ denote a precedence from task instance $\tau_i[n]$ to task instance $\tau_j[n']$. From the semantics of predefined operators, we have $\tau_i \overset{ops}{\rightarrow} \tau_j \Rightarrow \forall n,\ \tau_i[n] \rightarrow \tau_j[g_{ops}(n)]$, with g_{ops} defined as follows:

$$g_{*\hat{}k.ops}(n) = g_{ops}(kn) \qquad g_{/\hat{}k.ops}(n) = g_{ops}(\lceil n/k \rceil)$$
$$g_{\sim>q.ops}(n) = g_{ops}(n) \qquad g_{\text{fby}.ops}(n) = g_{ops}(n+1)$$
$$g_{\epsilon}(n) = n$$

The precedence relation is an over-approximation of the data-dependency relation. Indeed, there is a data dependency between $\tau_i[n]$ and $\tau_j[n']$, meaning that $\tau_j[n']$ consumes the data produced by $\tau_i[n]$, if and only if $\tau_i \overset{ops}{\rightarrow} \tau_j \wedge n' = g_{ops}(n) \wedge g_{ops}(n) \neq g_{ops}(n+1)$.

We can then adapt the encoding to our context. For each precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$, we must adjust the release dates and deadlines of each instance $\tau_i[n]$ such that $R_i^*[n] \leq R_j^*[g_{ops}(n)]$ and $D_i^*[n] \leq D_j^*[g_{ops}(n)] - C_j$. Concerning release dates,

we can easily prove that thanks to the synchronous semantics we already have $R_i[n] \leq R_j[g_{ops}(n)]$, so release dates do not need to be adjusted. Concerning deadlines, we need to transpose the formulae to relative deadlines to fit our task model. From the definition of relative deadlines: $D_i^*[n] \leq D_j^*[g_{ops}(n)] - C_j \Leftrightarrow d_i[n] \leq d_j[g_{ops}(n)] + r_j + g_{ops}(n)T_j - r_i - nT_i - C_j$.

Deadline Calculus In practice we do not need to unfold extended precedences to perform their encoding. Instead, we represent the sequence of deadlines of the instances of a task as a finite repetitive pattern called *deadline word*. A *unitary deadline* specifies the relative deadline for the computation of a single instance of a task. It is simply an integer value d. A *deadline word* defines the sequence of unitary deadlines for each instance of a task. The set of deadline words is defined by the following grammar: $w ::= (u)^\omega \quad u ::= d \mid d.u$. Term $(u)^\omega$ denotes the infinite repetition of word u. In the following, $w[n]$ denotes the n^{th} unitary deadline of deadline word w ($n \in \mathbb{N}$).

Let w_i denote the deadline word of task τ_i. A precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$ is encoded by a constraint relating w_i to w_j of the form:

$$w_i \leq W_{ops}(w_j) + \Delta_{ops}(T_i, T_j) - C_j + r_j - r_i$$

where, for all n, $W_{ops}(w_j)[n] = w_j[g_{ops}(n)]$ and $\Delta_{ops}(T_i, T_j)[n] = g_{ops}(n)T_j - nT_i$. Let $\psi_{ops}(\tau_j) = W_{ops}(w_j) + \Delta_{ops}(T_i, T_j) - C_j + r_j - r_i$.

Property 1. $\Delta_{ops}(T_i, T_j)$ is periodic and can be represented as a deadline word. The set of deadline words is closed under operation W_{ops} and under deadline words addition. As a consequence, $\psi_{ops}(\tau_j)$ is a deadline word.

Proof. By induction.

Property 2. The deadline words of a task graph $g = (V, E)$ can be computed with complexity $\mathcal{O}(|V| + |E| * |w_{max}|)$, where w_{max} denotes the deadline word which has the longest size in the task graph.

Proof. A reduced task graph is a DAG (when we do not consider delayed precedences), so we can compute the deadline words of the graph by performing a topological sort working backwards (starting from the end of the graph). As the complexity of a topological sort is $\mathcal{O}(|V| + |E|)$, the complexity of the algorithm is $\mathcal{O}(|V| + |E| * |w_{max}|)$ where w_{max} denotes the longest deadline word in the task graph.

For instance, for the Flight Control System program, we take $C_{PA} = 1$, $C_{AA} = 1$, $C_{FL} = 3$, $C_{PF} = 4$, $C_{PL} = 6$, $C_{NL} = 20$, $C_{NF} = 5$. To simplify, we take null durations for sensors and actuators. The result of the deadline calculus is: $w_{PA} = (10)^\omega$, $w_{AA} = (5.10.10.10)^\omega$, $w_{FL} = (9.10.10.10)^\omega$, $w_{PF} = (9)^\omega$, $w_{PL} = (15)^\omega$, $w_{NL} = (120)^\omega$, $w_{NF} = (100)^\omega$. The deadline words of tasks `AA` and `PA` state that, each first repetition out of four successive repetitions the two tasks have a shorter deadline as `PF` and `PL` execute. Notice that if we set deadline 5 for all the instances of `AA` (instead of a deadline word), this example is not schedulable.

5 Communication Protocol

As task precedences are encoded in task deadlines, inter-task communications do note require synchronization primitives (like semaphores for instance). Indeed, as long as tasks respect their deadlines, data is produced before being consumed, so tasks simply read from and write to some communication buffers allocated in a global shared memory when they execute. However, to respect the synchronous semantics, the input of a task must not change during its execution. Therefore, we propose a communication scheme, which ensures that the input of a task remains available until its deadline.

For a precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$, data produced by $\tau_i[n]$ may be consumed by $\tau_j[g_{ops}(n)]$ after $\tau_i[n+1]$ has started. This is illustrated in Fig. 6, which shows the schedule of two tasks related by extended precedences. Vertical lines on the time axis represent task periods and marks represent task preemptions. An arrow from A at date t to B at date t' means that task B may read at time t' from the value produced by A at time t. In Fig. 6(a), 6(c) and 6(d), when $A[1]$ executes, it must not overwrite the data produced by $A[0]$ because it is consumed by $B[0]$ and $B[0]$ is not complete yet. Therefore, we need a buffer to keep the value of $A[n]$ after $A[n+1]$ has started. In Fig. 6(b), the same data is consumed several times but $A[1]$ can freely overwrite the data produced by $A[0]$, so no specific communication scheme is required.

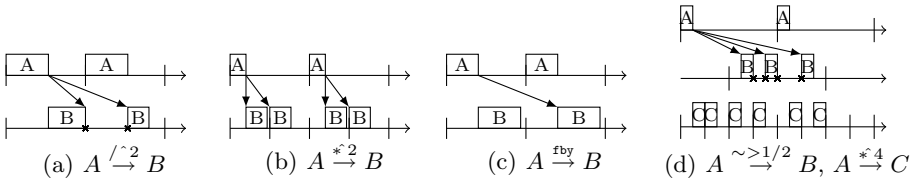


Fig. 6. Communications for extended precedences

The communication protocol allocates a buffer for each extended precedence of the graph. The producer of the data writes to the buffer only if $g_{ops}(n) \neq g_{ops}(n+1)$ (see the definition of data-dependencies in Sect. 4.3). The consumer simply reads from this buffer each time it executes. We allocate a double-buffer when the precedence contains a `fby` or a $\sim>$, to keep the previous and the current value of the data. This communication scheme is illustrated in more details in Sect. 6.1. It is of course not optimal because in many cases when we consider a set of precedences from a single task τ_i to several tasks τ_j we can use the same communication buffer for some of the tasks τ_j. Optimization will be treated in future work, we could for instance adapt the communication scheme proposed by [15] to our language.

6 Code Generation

The compiler generates C code with calls to the real-time primitives defined in the real-time extensions of the POSIX standard (POSIX.13 [13]). The code

generation can easily be adapted to any real-time operating system that provides dynamic priority scheduling. Each task is translated into a thread and the threads are executed concurrently by an EDF scheduler modified to handle deadline words.

6.1 Task Code Generation

The generated code consists of a single C file. The file starts with the declaration of one global variable for each communication buffer. For instance, for the communication from `PF` to `PL` (`acc_o` before graph expansion), we have the declaration: `int PF_o_PL_i1` (named after the output of the producer and the input of the consumer). For the communication from `PL` to `FL`, we have the declaration: `int PL_o_FL_i2[2]`, as there is a delay between the two tasks.

The file then contains a function for each task of the task set. The function mainly consists of an infinite loop, that wraps the function of the corresponding imported node with the code of the communication protocol. One step of the loop corresponds to the execution of one instance of the task. Once buffer updates are complete, the function signals to the scheduler that the current task instance completed its execution so that it can schedule another task.

For instance, the function for task `PL` is given in Fig. 7. The value returned by the external function `PL` is a single integer so we can directly assign its return value to an integer variable. When the external function returns a tuple, the output value is returned as a `struct` pointer in the parameters of the function. The variable `NL_o_PL_i3` is the communication buffer for precedence $NL \overset{\text{fby}\,.*\hat{}3}{\rightarrow} PL$. It is an array of size 2 as the precedence contains a delay. The delay is initialised at the beginning of the function. `PL` alternatively reads from value 1 and from value 0 of the array, starting with value 1 (`NL_o_PL_i3[(instance+1)%2]`). `PL` then copies its output value to the communication buffer `PL_o_order` for precedence $PL \rightarrow order$. Instruction `invoke_scheduler(0)` signals the completion of the task instance.

```
void *PL_fun(void * arg) {
  NL_o_PL_i3[1]=0; int instance=0;
  while (1) {
    PL_o = PL(FL_o_PL_i1,PF_o_PL_i2, NL_o_PL_i3[(instance+1)%2]);
    PL_o_order=PL_o;
    instance++;
    invoke_scheduler (0);
  }
}
```

Fig. 7. Code generated for task `PL`

The functions for tasks `FL` and `NL`, which produce data used by `PL` are given in Fig. 8. Variable `FL_o_PL_i1` is the communication buffer for precedence $FL \overset{/\hat{}4}{\rightarrow} PL$. It is updated once every 4 iterations, (`update_FL_o_PL_i1[instance%4]`). For precedence $NL \overset{\text{fby}\,.*\hat{}3}{\rightarrow} PL$, `NL` alternatively copies its output value to the value 0 and to the value 1 of the buffer `NL_o_PL_i3`.

```
void *FL_fun(void * arg) {
  int update_FL_o_PL_i1[4]={1,0,0,0}; int instance=0;
  while (1) {
    FL_o = FL(angle_FL_i1);
    if(update_FL_o_PL_i1[instance%4])
      FL_o_PL_i1=FL_o;
    instance++;
    invoke_scheduler (0);
  }
}
void *NL_fun(void * arg) {
  int instance=0;
  while (1) {
    NL_o = NL(NF_o_NL_i1,pos_r_NL_i2);
    NL_o_PL_i3[instance%2]=NL_o;
    instance++;
    invoke_scheduler (0);
  }
}
```

Fig. 8. Code generated for tasks `FL` and `NL`

The main function then creates one thread for each task, initializes the EDF scheduler and attaches the threads to it. For instance, the thread for task `PL` is created by the following function call: `pthread_create(&tPL,&attrPL,PL_fun, NULL)`. `tPL` is the thread created for this task. `attrPL` contains the real-time attributes of the task. `PL_fun` is the function executed by the thread. The last parameter `NULL` stands for the arguments of `PL_fun`.

6.2 Implementing EDF with Variable Deadlines

We choose to prototype the scheduler using MaRTE Operating System [14], which was designed specifically to ease the implementation of application-specific schedulers while remaining close to the POSIX model. We modify the EDF scheduler provided with the OS to support deadline words. To summarize, the EDF scheduler is defined as a high-priority thread created by the main function of the file. The scheduler thread is itself scheduled by the kernel of the OS. It becomes active only when scheduling actions must be taken, which is when the following *scheduling events* (implemented by means of signals) occur: the current instance of a task completes its execution or a new task instance is released. The scheduler then computes the most urgent task among the ready tasks (tasks released and not complete yet), resumes the execution of the corresponding thread where it stopped and suspends the execution of the currently executing thread, if any. The scheduler thread then becomes inactive until the next scheduling event.

The support of deadline words requires very few modifications. We define deadline words and modify the structure describing task real-time attributes as shown in Fig. 9. Then, we modify the function that programs the next instance of a task when the current instance completes. For a task τ_i, the attributes of which are described by the value `t_data`, the release date and the deadline of its new instance are computed as described in Fig. 10 ($D_i[n] = R_i[n] + d_i[n]$). The function `incr_timespec(t1,t2)` increments `t1` by `t2` and the function `(t1,t2,t3)` sets `t1` to `t2+t3`.

We can see that the overhead due to the support of deadline words remains very reasonable. Altogether, we modified about 20 lines of code of the original

```
typedef struct dword { struct timespec *dds; int wsize; } dword_t;
typedef struct thread_data {
  struct timespec period;
  struct timespec initial_release;
  dword_t dword;
  struct timespec next_deadline;
  struct timespec next_release;
  int instance;
} thread_data_t;
```

Fig. 9. Data type representing task real-time attributes

```
dword_t dw = t_data->dword; t_data->instance++;
incr_timespec (&t_data->next_release,&t_data->period);
add_timespec (&t_data->next_deadline,&t_data->next_release,
              &(dw.dds[t_data->instance%dw.wsize]));
```

Fig. 10. Releasing a new task instance

EDF scheduler, which is 300 lines of code long. We compiled and executed the C code generated for the Flight Control System and it behaved as expected.

7 Related Works

The language used in this article relies on a specific class of clocks to handle the multi-periodic aspects of a system. Real-time periodic clocks have also been introduced in [3], but they do not include clock transformations to efficiently handle rate transitions. Our rate-transition operators are also very similar to the rate transition blocks of SIMULINK [10]. Yet, as far as we know, for models using such blocks, the code generation tool (Real-Time Workshop) relies on a Rate-Monotonic scheduler [9] used with semaphores to handle task communications, which is not an optimal scheduling policy for this scheduling problem.

The scheduling of *multi-rate* Synchronous Data Flow (SDF) is a well studied problem (see for instance [8, 16, 11]). In particular [16] studies the implementation of SDF with a dynamic scheduler using preemption. However, though multi-rate systems are at the chore of SDF graphs, SDF operations are *not periodic*, they are not released periodically and their relative deadline is not their period. As a consequence, these results do not apply to our problem.

[15] deals with the execution of a set of synchronous tasks, the semantics of which is very close to our task sets, with a dynamic scheduler. However, the authors do not detail how the synchronous task set is obtained, for instance how it is translated from a synchronous language, and task precedences are not specified in the task set.

A simple solution to the problem of scheduling tasks related by extended precedences is to unfold the extended precedence graph on the hyperperiod of the tasks and use [2] to encode the simple precedences of the unfolded graph. This solution replaces each task τ_i by HP/T_i duplicates (where HP is the hyperperiod of the tasks) in the unfolded graph. This can lead to important computation overhead at execution as the scheduler needs to make its decisions according to a task set that will contain many tasks. Our solution does not duplicate any task, so the scheduler takes less time to make its decisions.

8 Conclusion

We proposed a language for programming critical systems with multiple real-time constraints, along with its compiler, which automatically translates a program into a set of independent real-time tasks programmed in C with POSIX.13 real-time extensions. The generated code is schedulable optimally by a slightly modified EDF scheduler and requires no synchronization primitives. Though tasks are scheduled concurrently and preemptions are allowed, the generated program respects the real-time and the functional semantics of the original program.

References

1. A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. *Sci. of Compu. Prog.*, 16(2), 1991.
2. H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2, 1990.
3. A. Curic. *Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints*. PhD thesis, Université Joseph Fourier, Grenoble, 2005.
4. J. Forget. Programming and implementing Control-Command systems, progression report 4. Technical Report RT4/12144, ONERA, 2009. *Under publication. For reviewing purposes*: www.cert.fr/anglais/deri/jforget/RT4-12144.pdf.
5. J. Forget, F. Boniol, D. Lesens, and C. Pagetti. A multi-periodic synchronous data-flow language. In *11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, Nanjing, China, Dec. 2008.
6. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proc. IEEE*, 79(9), 1991.
7. N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*, Passau, Germany, 1991.
8. E. Lee and D. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transaction on Computer*, C(36), 1987.
9. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.
10. The Mathworks. *Simulink: User's Guide*.
11. H. Oh, N. Dutt, and S. Ha. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *Asia and South Pacific Design Automation Conference (ASP-DAC'06)*, Yokohama, Japan, Jan. 2006.
12. B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, USA, 2002.
13. POSIX.13. *IEEE Std. 1003.13-1998. POSIX Realtime Application Support (AEP)*. The Institute of Electrical and Electronics Engineers, 1998.
14. M. A. Rivas and M. G. Harbour. POSIX-Compatible Application-Defined Scheduling in MaRTE OS. In *14th Euromicro Conference on Real-Time Systems (ECRTS'02)*, Washington, USA, 2002.
15. S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4), 2005.
16. D. Ziegenbein, J. Uerpmann, and R. Ernst. Dynamic response time optimization for SDF graphs. In *IEEE/ACM international conference on Computer-aided design (ICCAD'00)*, San Jose, USA, Nov. 2000.

Re-verification of a Lip Synchronization Protocol using robust reachability

Piotr Kordy
kordy@cs.utwente.nl
Formal Methods and Tools

Rom Langerak
langerak@cs.utwente.nl
Formal Methods and Tools

Jan Willem Polderman
j.w.polderman@math.utwente.nl
Mathematical Systems and Control Theory

University of Twente, Drienerlolaan 5, 7522 NB Enschede, The Netherlands

The timed automata formalism is an important model for specifying and analysing real-time systems. Robustness is the correctness of the model in the presence of small drifts on clocks or imprecision in testing guards. A symbolic algorithm for the analysis of the robustness of timed automata has been implemented. In this paper, we re-analyse an industrial case lip synchronization protocol using the new robust reachability algorithm. This lip synchronization protocol is an interesting case because timing aspects are crucial for the correctness of the protocol. Several versions of the model are considered: with an ideal video stream, with anchored jitter, and with non-anchored jitter.

1 Introduction

Timed automata [2] is a widely used and successful formalism to analyse real-time systems. Timed automata are automata extended by clock variables that can be tested and reset. Numerous real-time systems have been specified and analysed by the tool UPPAAL [5, 17] and the approach can be said to be mature and industrially applicable.

However, if we want to implement a system then robustness becomes an issue. We need to know if the system is resilient with respect to small perturbations. Timed automata in its original form may be crucially dependent on perfect precision of the clocks. Therefore, several publications have suggested alternative semantics of timed automata that take into account perturbations. In particular, the skewed clocks automata from [1] can have arbitrary rates for clocks, the "tube languages" from [9, 11] deal with open sets of trajectories, "perturbed" timed automata from [14] are subjected to an infinitesimal noise, and the implementable timed automata from [19, 20] should be implementable using discrete clocks.

Semantics can only be said to be successful if it can be applied in practice. In this paper we are interested in the work that was initiated by Puri [13]. He considered drifting clocks and showed that timed automata models are not robust with respect to safety properties, meaning that a model proven to be safe under the standard ideal semantics might not be safe even if clocks drift by an arbitrarily small amount. The region based algorithm has been proposed to calculate set of states that are reachable for *any* clock drift. Puri's approach has been extended by the introduction of the concept of stable zone [7], which made it possible to implement an efficient algorithm that can be used in practice.

To check the new performance of the new algorithm the best way is to apply it to an industrial case study. On the UPPAAL homepage [17] a number of case studies can be found in which the UPPAAL tool has been applied. We investigated several of them and we have chosen the case study where a lip synchronisation algorithm is analysed. This algorithm is used to synchronize multiple information streams sent over a communication network, in this case audio and video streams of a multimedia application. We chose this case study mainly because timing is an important aspect of synchronisation and therefore this algorithm can be expected to be sensitive to small disturbances of the clock drift.

Submitted to:
FMA 2009

Structure of the paper The rest of the paper is organised as follows. Section 2 introduces the lip synchronisation problem. Section 3 presents the modelling formalism and tool used in the analysis of the lip synchronisation protocol. Section 4 provides a description of a model of the lip synchronisation protocol. Section 5 presents the verification results and Section 6 gives a concluding discussion.

2 Lip Synchronisation Problem

The problem of lip synchronisation has been present in the literature [16, 4]. Here we present it briefly, for more detailed description look in [6]. In this paper, we consider the problem of synchronising of audio and video streams. We consider a scenario when audio and video are transmitted as separate streams that need to be synchronised at the sink.

The overview of the basic configuration can be seen in Figure 1. There are two stream sources: one for sound and one for video. These streams arrive at a presentation device. We need to ensure that both streams play synchronised within certain level of tolerance. This is the problem that lip synchronisation protocol addresses.

The protocol is implemented using several components: *sound* and *video* managers, and a *controller*. Communication between components is done using signals. When presentation device receives sound packet it sends a *savail* [1] signal to the *Sound Manager*. At an appropriate moment the Sound Manager sends *spresent* signal to the Presentation Device to indicate that the packet should be played. The *Video Manager* has similar behaviour and uses signals *vavail* and *vpresent*. The *Controller* contains the main body of the protocol. It receives signals *sready* and *vready* from the managers. The signals indicate that sound and/or video packets can be presented. The Controller decides if it is the correct time to play the packet. Confirmation is done using *sok* and *vok* signals, respectively. If it is not possible to synchronise, the Controller signals an error and enters an error state.

The requirements for acceptable synchronisation between the two streams are the following:

- The time granularity is 1 millisecond.
- A sound packet is presented every 30 milliseconds and no jitter is allowed.
- Optimally, a video packet should be presented every 40 milliseconds. However, we allow some margin of error:
  - video frames may precede sound frames up to 15 milliseconds and may lag up to 150 milliseconds.
  - we allow a 5ms jitter that is, a video package may be no less than 35 ms late and no more than 45 ms away from ideal presentation time (Anchored Jitter) or from previous packet (Non-anchored Jitter)

3 Modelling Formalism

3.1 Timed Automata

For our purposes we use the existing timed automata model from [6]. The modelling formalism is based on a network of timed automata. The network of timed automata consists of the parallel composition

[1] names are usually prefixed with s for sound and v for video

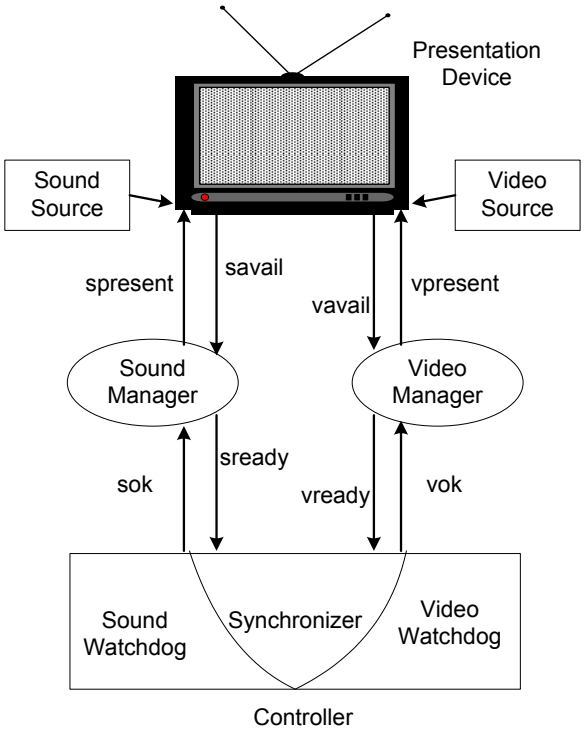


Figure 1: Overview of the structure of the lip synchronisation system

of a number of timed automata, and a configuration. A timed automaton is an automaton consisting of locations and edges that are extended with real valued variables called clocks.

Edges and locations are labelled. The labels of an edge may consists of several optional components: a guard, a synchronisation label, a set of clocks to reset, and assignments to integer variables. The guard on clocks and/or on data variables expresses under which conditions we are allowed to take a transition. If there is no guard, the condition is interpreted as true. Because later we consider a so called robust semantics, we limit guards on clocks to the form : $x \leq c$ or $x \geq c$ where x is a real valued clock and c is natural number. We also allow a formula that is a combination of the above terms using logical *and*. When we take a transition we may perform a synchronisation. The synchronisation label must be synchronised with its counter part. The synchronisation rules are similar as in CSS [12]. When there is no synchronisation, a label is interpreted as an internal action (similar to τ-actions).

The labels of locations consist of the name of a location, an optional invariant, and can be marked as *committed* or *urgent*. An invariant is a constraint on clocks, indicating how long we can stay in a location. It is similar to guard but only upper bound constraints are allowed. When a location is marked as committed, we have to leave this location without any delay or any interleaving actions. This is useful to ensure atomicity of sequence of transitions. When a location is urgent time is not allowed to pass in that location.

The configuration consists of the names of timed automata composing the system, global variables, and channels. The synchronisation happens through channels and synchronisation labels are names of channels. Channels can be *urgent*. When a channel is urgent, we have to take that transition as soon as possible that is without delay. There can be no guards on edge with urgent channel.

The state of timed automaton is of the form $(\bar{q}, v)$ where $\bar{q}$ is a control vector and v is the clock

valuation. The control vector shows the current location for each timed automata in the network and the clock valuation indicates value of each clock and integer variables. The initial state consists of initial locations for each timed automaton and all clocks and variables equal to value 0. From a state it is possible to take two types of transitions: *delay* and *edge* transition. When we take a delay transition, all clocks are progressing at the same speed within the values allowed by the location invariants. An edge transition can be internal or a synchronisation. An internal transition can occur when the network is at the location in which it can take an edge with no synchronisation label. The guard must be satisfied by the current clock valuations. A synchronisation transition occurs when two edges can synchronise over complementary synchronisation labels. Guards of both edges must be satisfied.

3.2 Model Checking

The continuous time leads to infinitely many states. Fortunately, as noted in [2] similar states can be grouped into *regions*. However region automaton is not the most efficient representation of the state space of a timed automaton. It suffers from a combinatorial state explosion, which is dependent on the size of the constants used. *Zones* are used as a more efficient representation of the state space [8, 10, 21] as they represent the state space in a more aggregated way. In the tool UPPAAL [5] more effective zone based algorithm is used.

The UPPAAL tool is able to check for reachability properties. Those properties are of the form:

$$\varphi ::= \forall\Diamond\beta \mid \exists\Box\beta \qquad\qquad \beta ::= a \mid \beta_1 \wedge \beta_2 \mid \neg\beta \mid \beta_1 \Rightarrow \beta_2$$

where a is an atomic formula being either an atomic clock (or data) constraint or a component location $(A_i \text{ at } l)$. Atomic clock (data) constraints are integer bounds on individual clock (data) variables (e.g. $1 \leq x \leq 3$).

Intuitively for $\forall\Box\beta$ to be satisfied all reachable states must satisfy β. For $\exists\Diamond\beta$ to be satisfied some reachable state must satisfy β.

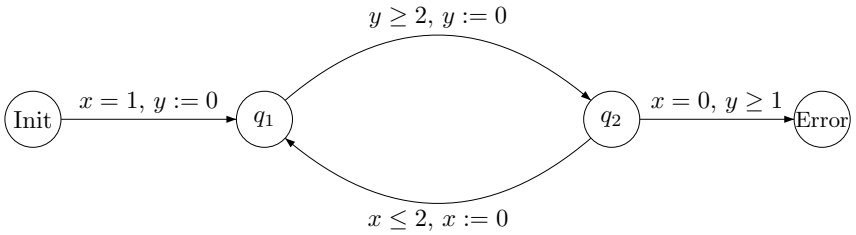
3.3 Robustness Problem

Clocks in a timed automata network are synchronous. Puri [14] has shown that this assumption is not robust to even infinitely small clock drifts. In short, it means that we can reach states that are not reachable in normal semantics for *any* value of the clock drift. He proposed new reachability semantics for timed automata and we will call it the *robust semantics*. The idea is to have a parametrised reachability, where a parameter defines how much clocks are allowed to drift. In normal reachability, when time progresses by t time units, the new clock valuation is $v+t$. When we allow clock drift parametrised by ε, the new clock valuation can have ε small differences between clocks. Formally

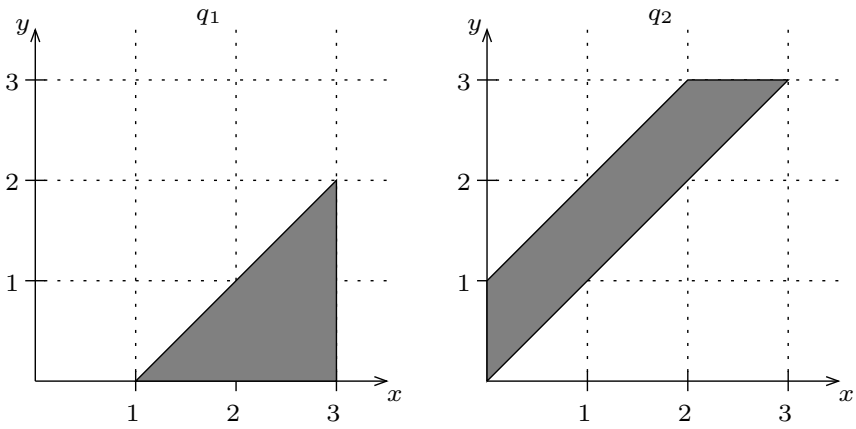
$$v'(x_i) - v(x_i) \in [(1-\varepsilon)t, (1+\varepsilon)t], \text{ for } i = 1, \ldots, n$$

where n is number of clocks, v' is a valuation after the time transition and v is a clock valuation before the time transition. Let $\mathsf{Reach}_\varepsilon(s_0)$ denote the reachable set of states from the initial state s_0 in the robust semantics. Unfortunately parametric model checking of timed automata with three clocks and only one parameter is known to be undecidable [3, 18]. What Puri proposes is the reachability when the drift ε is infinitely small. Formally

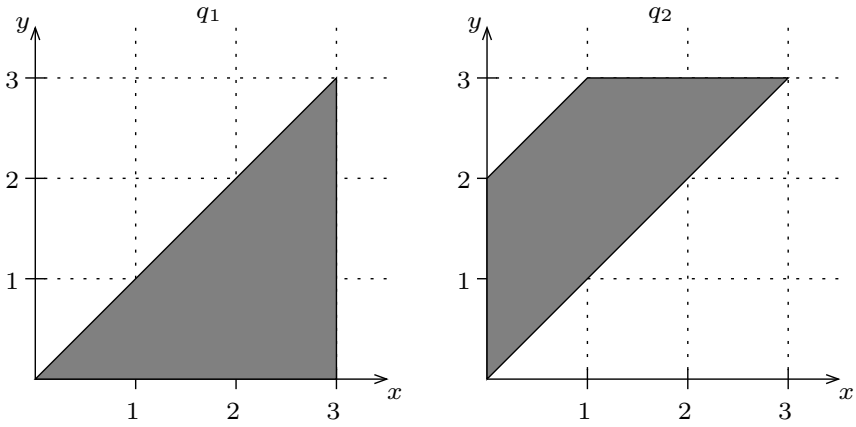
$$R_{\varepsilon\to 0}(s_0) = \bigcap_{\varepsilon>0} \mathsf{Reach}_\varepsilon(s_0)$$



(a) A timed automaton



(b) Reachable set of states in the normal semantics



(c) Reachable set of states in the robust semantics

Figure 2: Example of timed automaton with different reachability under normal and robust semantics.

Puri shows that calculating $R_{\varepsilon\to 0}(s_0)$ is decidable. He proposes a region based algorithm.

It may seem that $R_{\varepsilon\to 0}(s_0)$ and $\mathsf{Reach}(s_0)$ are the same since ε is small, but this is not the case. Consider the example shown in Figure 2(a). This timed automaton has two clocks x and y and four locations. From location Init we can only go to location q_1 and the value of the clocks will be $x = 1$ and $y = 0$. In the precise semantics, following the cycle between locations q_1 and q_2, we will get the reachable set of states $\mathsf{Reach}(\mathscr{S})s_0$ depicted in Figure 2(b). We want to avoid Err location. The Err location is not reachable for both $\alpha = 2$ and $\alpha = 3$.

Now we consider the case for the robust semantics. Let e_1 be edge from location q_1 to q_2 and e_2 edge from location q_2 to q_1. Notice that for any $0 \le \beta \le 1$ the following sequence of transitions is possible: $(q_1; x = \beta, y = 0) \overset{2-\beta}{\to}_{\varepsilon} \overset{e_1}{\to}_{\varepsilon} (q_2; x = 0, y = 2 - \beta + \varepsilon) \overset{\beta-\varepsilon}{\to}_{\varepsilon} \overset{e_2}{\to}_{\varepsilon} (q_1; x = \beta - \varepsilon, y = 0)$. Hence, if we cycle sufficient number of times for any $\varepsilon > 0$ we can reach state $(q_1; x = 0, y = 0)$ which is not reachable in the normal semantics $\mathscr{S}$. Thus, the Err location is robustly reachable for $\alpha = 2$. This shows that the normal semantics is not robust with respect to small clock perturbations. Even small changes in the clock drift may lead to a dramatic change in the behaviour of a system. We avoid the Err location only in the normal semantics, but not in the robust semantics. We say that such safety property is non-robustly satisfied. If a system has non-robustly satisfied property it is not implementable because its correctness depends on the mathematical idealization of the normal semantics.

3.4 Verification Tool

Puri proposed an algorithm to calculate $R_{\varepsilon\to 0}(s_0)$ that is based on regions. Basically the algorithm finds regions that are on the cycle in the region graph. Such regions have the property that we can drift form any point to any other point in that region in robust semantics. When a part of such a region is encountered in the search, the whole region is added to the reachable set of states. In [7] the notion of a *stable zone* has been introduced. Basically the stable zone has the same property as the region on the cycle that is, we can drift from any point in the stable to any other point in that stable zone. Thus, this is a good starting point for the zone based algorithm to calculate robust reachability. Together with the Aalborg University the prototype tool is being developed based on UPPAAL 4.1.1 and it is used in this paper.

4 The Model

In this section UPPAAL model is presented. It follows the specification given in [6] which in turn was derived from the specification given in LOTOS [15]. The model represents the specification of the video and sound managers and the synchroniser from Figure 1.

The model is shown in Figure 3. Sound and Video Managers are modelled by automata *VideoMgr*, *SoundMgr*, *VideoWdg*, *SoundWdg* and *UrgMon*. The Synchronizer consists of *Synch*, *VideoSynch*, *SoundSynch* and *SoundClock*. The external environment that is the sound and video streams are modelled by *VideoStr* and *SoundStr*. We briefly discuss the components.

The stream managers Both stream managers are quite simple. After receiving a signal *vavail* or *savail* indicating that the video or sound packet is available, they forward the signal immediately to the synchroniser using *vready* and *sready*. The immediacy is ensured by marking the locations *vm2* and *sm2* as committed. Next the manager waits for a confirmation from the controller (the confirmation signal comes from the watchdogs) meaning that the packet can be played. This is done using signals *vokk* and

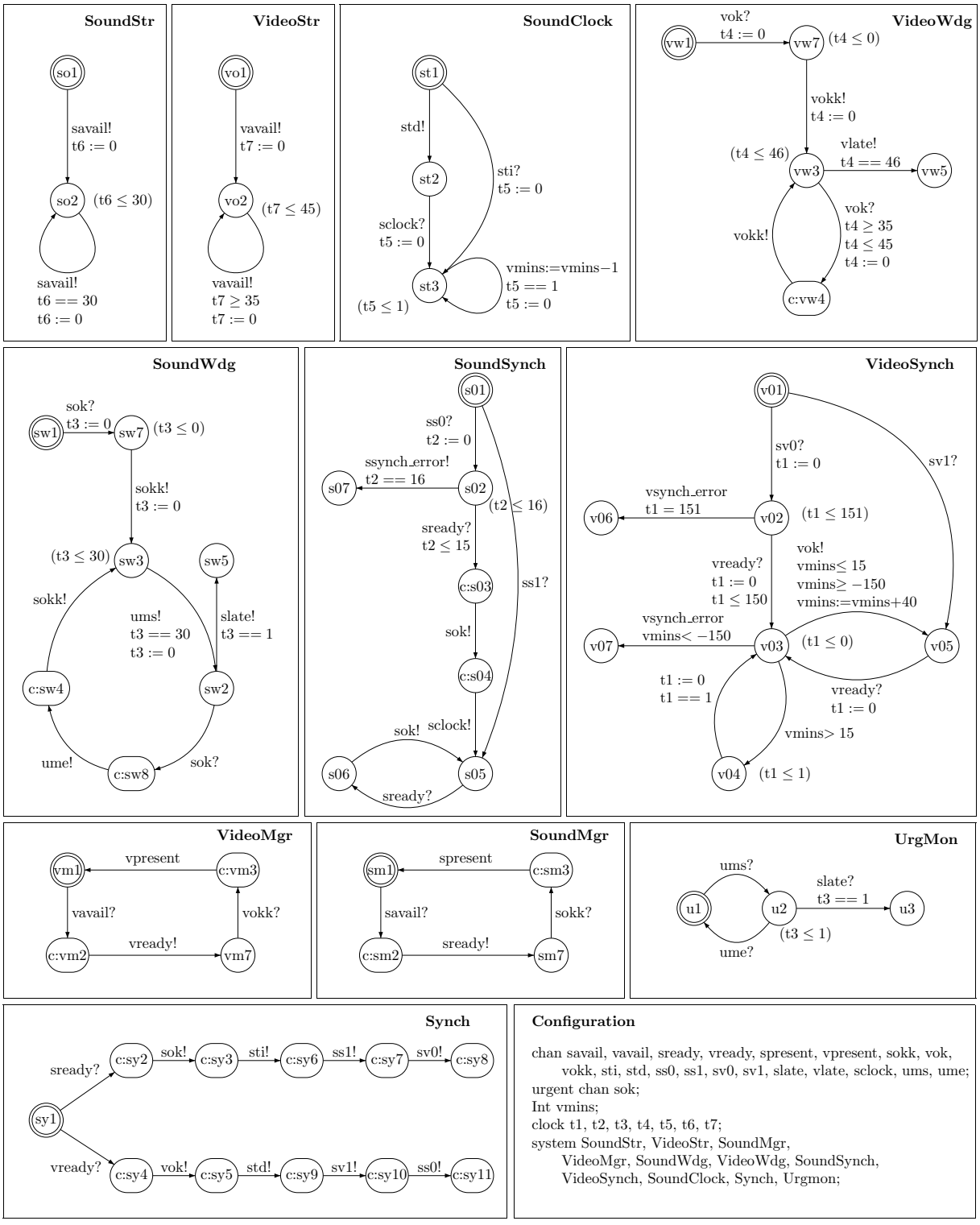


Figure 3: A model of Lip Synchronisation Protocol

sokk. The confirmation is immediately forwarded to the presentation device using signals *vpresent* and *spresent*. Since the presentation device is not modelled, those actions are internal.

The watchdog timers The role of watchdog timers is to ensure that the time between presentations of subsequent media packets is within certain time bounds. We discuss the video watchdog. The timing requirement is that consecutive video packets are played between 35 ms and 45 ms. Initially the watchdog waits for the first packet to arrive (signal *vok*) and sends immediately the confirmation to the video manager using signal *vokk*. We ensure that no time passes between *vok* and *vokk* signals. The signals *vok* and *vokk* constitute in a way a complex signal that allows synchronisation between *VideoSynch*, *VideoMgr* and *VideoWdg*. After presenting the first packet the time is measured until the next packet arrives. In order to ensure proper timing of the presentation of the packet, the transition leaving location vw3 is guarded by $35 \leq t4 \leq 45$. If *vok* does not occur before 45 ms passes, *vlate* error is given.

The *SoundWdg* is slightly more complicated because we must ensure that sound packets are played exactly every 30 ms. Similarly to the video watchdog it waits for the confirmation (signal *sok)* that the first packet should be presented and relays the signal to the *SoundMgr* using the signal *sokk* without time delay. After that the clock t3 is used to measure the time between consecutive presentations of the sound packets. To ensure that exactly 30 ms passes between sound packets, *UrgMon* is used and signal *sok* is marked as urgent. If a sound packet is not available in 30 ms the *slate* error is generated.

The synchroniser The role of a *Synch* is to initialise the other automata. Depending on whether a sound or a video packet arrives first, automata can be initialised in two ways. If signal *vready* or *sready* arrives then we confirm that the packet can be presented (signal *vok* or *sok*) and initialise *SoundClock* (signal *std* or *sti*) then initialise *VideoSych* (signal *sv1* or *ss1*) and at last we initialise *SoundSynch* (signals *sv0* or *ss0*). Note that all locations except sy1 are committed to ensure that initialisation is done immediately.

The sound clock The *SoundClock* is a discrete clock that ticks every millisecond. It is started at the moment the first sound packet arrives. It can be initialised by signal *sti* if the sound packer is first or by combination of *std* and *sclock* signals if video packet arrives first.

The clock is used to compute the skew between sound and video streams. The skew is stored in *vmins* variable. Every time the clock ticks it is decreased by one.

The sound synchroniser The *SoundSynch* can be initialised it two ways. If a sound packet is first it receives *ss1* signal and starts the repeating behaviour immediately. If a video packet is first then it checks if there is a synchronisation error - it can happen only when the first sound packet does not arrive within 15 ms after the first video packet. This is the requirement of the lip synchronisation. After the fist sound packet arrives it initialises *SoundClock* through the *sclock* signal and starts the repeating behaviour

The repeating behaviour is very simple. If it receives signal *sready* that the sound packet has arrived, it send a signal *sok* indicating that it can be presented.

The video synchroniser The *VideoSynch* is quite complex. Similarly as *SoundSynch* it can be initialised in two ways. If the video packet arrives first it goes immediately to the repeating behaviour through signal *sv1*. If sound packet arrives first, it checks if a video packet is received within 150 milliseconds but not earlier than 15 milliseconds from the sound packet. If more than 150 milliseconds have passed then *vsynch_error* is generated.

In the repeating behaviour, *VideoSynch* checks if there is too much skew between sound and video packets. After receiving *vready* signal, it checks the lip synchronisation requirement immediately (the t1$\leq$ 0 invariant). Now we have three possibilities:

- The video presentation is more than 150 milliseconds later than the sound presentation. This is true when *vmins* is less than -150. In such case *vsynch_error* is generated.
- The video is more than 15 ms early with respect to the sound packet. This is the case if *vmins*$>$ 15. In such case the presentation of the video frames are postponed. We enter a state where we are forced to wait one millisecond. After that we check the synchronisation requirement again.
- The video and the sound packets are sufficiently synchronised. In such case we send a signal *vok* to present a video packet and we update the *vmins* variable.

The media streams The informal specification of the protocol does not make any assumptions about the streams. Several possible streams are modelled and are further described in Section 5.

5 Verification

5.1 Verified properties

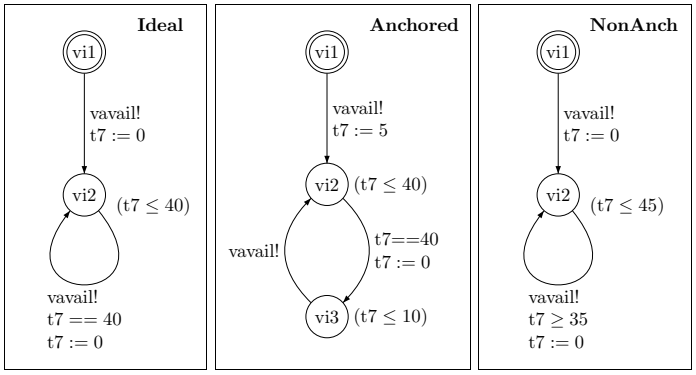


Figure 4: Three variations of video stream

We have followed [6] and we assumed that the sound stream is ideal and arrives every 30 ms. The perturbations may affect the video stream. There are three possible video streams that are investigated:

- An *ideal* video stream that delivers a video frame every 40 ms.
- A video stream with *anchored jitter* that have a rate of 40 ms and a variation of $\pm$5 ms.
- A video stream with *non-anchored jitter* where the variability between each two consequent frames is minimally 35 ms and maximally 45 ms.

Figure 4 shows automata representing different variations of the video stream.

Another variation that was investigated is the initial delay of video and sound streams. The first option is that the starting time of streams is left unspecified; the other possibility is that both streams start at the same time.

The verification is done using error location reachability. Each error location reachability was done using normal and robust semantics. The reachability properties are all of the form:

$$E\Diamond A.l \ \ and \ \ not\,(B_1.l_1 \ \ or \ \ldots \ B_n.l_n)$$

The answer to such a query will be positive if there exists a path in timed automata network which will eventually reach location l in A but all locations l_i in B_i will be avoided. The location l will be the error location we are checking. The second part is to ensure that timed automata network did not reach another error location as this might have caused another error location to be reachable. The following error location have been modelled and checked for reachability:

- Initial sound synchronisation error in the *SoundSynch* (location s07)
- Initial video synchronisation error in the *VideoSynch* (location v06)
- Video synchronisation error in the *VideoSynch* (location v07)
- Video late error in the *VideoWdg* (location vw5)
- Sound late error in the *SoundWdg* (location sw5)

5.2 Results

We have implemented the algorithm for the robust semantics in a prototype tool based on UPPAAL 4.1.1. The normal semantics reachability analysis is done using UPPAAL 4.1.1. Our implementation at best can be as good as a depth first search for the normal reachability. Thus all the results presented here are run using depth first search. Experiments were performed on a PC with an AMD 1.2 GHz processor with 768MB of RAM. In [6] the state space was reduced by marking the error locations as committed. We have not done this optimisation.

Error location	Ideal		Anchored		Non-anchored	
Init Sound Synch(s07)	T	0.5	T	0.1	T	0.2
Init Video Synch(v06)	T	1.5	T	56.7	T	55.7
Video Synch (v07)	F	3.3	T	57.9	T	26.5
Video Late (vw5)	F	3.3	T	0.2	F	55.5
Sound Late (sw5)	F	3.2	F	61.8	F	57.2
Init Sound Synch(s07*)	T	0.1	T	0.2	T	0.2
Init Video Synch(v06*)	T	7.4	T	4613.3	T	1260.1
Video Synch (v07*)	T	3378.2	T	3168.4	T	674.9
Video Late (vw5*)	F	5636.4	T	7.5	F	5834.5
Sound Late (sw5*)	F	5378.2	F	5591.2	F	5724.4

Table 1: Verification results for streams with possible initial delay for both normal and robust semantics (marked with *)

Table 1 gives the results of the verification of the lip-synchronisation protocol for the various reachability properties. In the leftmost column we have a type of error that can occur. In the case of the error location being not reachable we mark it with F and if the error location is reachable we put T. In the second column we have a verification time given in seconds.

We can see that for all kinds of video streams, the initial sound and video synchronisation errors can occur. This can be explained by the fact that video or sound stream can postpone sending packet. This allows for the gap between sound and video packet to be arbitrarily long and reaching error location.

The anchored and non-anchored video streams can encounter video synchronisation error and can reach location vw07. In both cases it is enough to wait as long as possible but avoiding initial video synchronisation error and then the gap between sound and video packet can be enlarged due to allowed jitter.

Only in the case of the video stream with anchored jitter video frames can be late. In anchored jitter maximal gap between two consecutive packets is 50 ms. This is 5 ms more than allowed gap thus video frames can be late.

Error location	Ideal		Anchored		Non-anchored	
Init Sound Synch(s07)	F	0.2	F	0.4	F	86.1
Init Video Synch(v06)	F	0.2	F	0.5	F	85.1
Video Synch (v07)	F	0.2	F	0.5	T	36.8
Video Late (vw5)	F	0.2	T	0.2	F	83.1
Sound Late (sw5)	F	0.2	F	0.5	F	81.8
Init Sound Synch(s07*)	F	81.8	F	153.2	F	178.4
Init Video Synch(v06*)	F	391.1	F	397.2	F	357.9
Video Synch (v07*)	T	275.4	T	293.7	T	244.2
Video Late (vw5*)	F	394.4	T	9.2	F	385.4
Sound Late (sw5*)	F	391.5	F	385.3	F	401.2

Table 2: Verification results for streams without initial delay for both normal and robust semantics (marked with *)

The results where both streams are forced to start at the same time are shown in Table 2. The presentation format is the same as in Table 2. In such a situation the initial synchronisation errors are not reachable. In the case of an ideal video stream we do not encounter any errors. In the case of an anchored video stream again the video can be late. The reason is the same as previously. The non-anchored video stream can lead to out of video synchronisation error. This is because the gap can accumulate over time.

The second part of Table 1 and Table 2 shows the result for the robust semantics. They are marked with * next to the location name. It is worth mentioning here that if some location is reachable in the normal semantics then it is reachable in the robust semantics. The main difference between normal and robust semantics is that a video synchronisation error is always possible for all kinds of video stream, with or without allowed initial delay.

We try to explain this for the case of ideal video stream without initial delay, as all other cases are less restrictive. The variable vmins is decreased every millisecond. The timing is provided by the clock t5. In the case of an ideal video stream, every 40 ms (ensured by clock t7) a video packet is sent. If vmins is small enough, so we do not have to enter location v04 and vmins is increased by 40. Thus over time vmins oscillates around the same base value. Now assume that clocks t5 and t7 desynchronise by ε every millisecond. For sufficiently many cycles the base value over which vmins oscillates can be changed up or down. If vmins is too large it will be remedied by visiting location v04, but no such mechanism exists when vmins is getting smaller. Finally we will reach v07 and video synchronisation error will occur. Other types of errors use the clocks that cannot accumulate the drift because their reset time is

synchronised with the signals. So the verification results from normal and robust semantics are the same.

The conclusion is that if we play video long enough we are not able to guarantee that the protocol will not desynchronise, no matter how precise clocks we have. We are only able to guarantee proper lip synchronisation for a playback with limited time.

Deadlocks In [6] in addition the deadlock detection is done. We do not do deadlock detection as current status of the theory does not allow the tool to detect deadlocks robustly. The main limiting factor is the fact that we do not allow the guards to be strict. To do the deadlock detection we need to detect when we reach state that cannot leave through any transition. For that we need to complement guards on the edges and that introduces strict inequalities. The brief manual analysis of the specification reveals that new deadlocks can be reached in robust semantics. For example non-anchored video can send video packet at 45 ms but because of slight desynchronisation clock t4 have value $45+\varepsilon$ thus edge leading to location vw4 is not enabled any more.

As a side note let us mention that in [6] authors report one deadlock and they expected the other deadlock that should be detected by the UPPAAL, but full state search did not revealed it. It appears that the reason for not detecting the deadlock must have been the early imperfection of the UPPAAL tool, as the current version 4.1.1 detect both deadlocks.

6 Conclusions

We have re-verified a lip synchronisation protocol using robust semantics. The original protocol has been previously model checked using UPPAAL [6] and has been presented in a number of different formalisms [16, 15, 4].

The verification results using robust semantics are slightly different from normal semantics. The choice of case study was done to maximise probability of different results so this result was anticipated. The robust reachability analysis allowed us to identify the problem with the lip synchronisation protocol. For a continuous playback we are not able to make the clocks precise enough to ensure that video and sound do not become desynchronised. The sound and video can stay synchronised only for a limited time, and this time is depending on the precision of the clocks.

The verification of the lip synchronisation protocol gave us also the possibility to evaluate the performance of the robust reachability algorithm. The verification of a lip synchronisation using robust semantics takes significantly more time than verification using normal semantics. The main reason is that the model uses the variable vmins as a kind of discrete clock which divides the state space into many small pieces. This forces our algorithm to add many stable zones, which is expensive.

The limitation of our algorithm is its inability to detect deadlocks. In the case of the industrial case study this is important feature. We believe that the ability to detect deadlocks would identify more problems - mostly connected to the way time-out is modelled. Another limitation is that it is not possible to use strict guards. In the context of robustness where we allow small clock drifts, we believe that differentiating between strict and non-strict guards is not an essential feature. Unfortunately this feature is needed for deadlock detection. This will be an important direction of our future work. At the moment only reachability properties can be analysed. Future research will investigate the extension of the algorithm with the possibility to check liveness properties.

References

[1] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger & Pei-Hsin Ho (1993): *Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems.* In: *Hybrid Systems, LNCS* 736. Springer, pp. 209–229.

[2] Rajeev Alur & David L. Dill (1994): *A theory of timed automata*. *Theoretical Computer Science* 126(2), pp. 183–235.

[3] Rajeev Alur, Thomas A. Henzinger & Moshe Y. Vardi (1993): *Parametric real-time reasoning*. In: *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, pp. 592–601.

[4] Ahmet F. Ates, Murat Bilgic, Senro Saito & Behçet Sarikaya (1996): *Using Timed CSP for Specification Verification and Simulation of Multimedia Synchronization*. *IEEE Journal on Selected Areas in Communications* 14(1), pp. 126–137.

[5] Johan Bengtsson & Wang Yi (2003): *Timed Automata: Semantics, Algorithms and Tools*. In: *Lectures on Concurrency and Petri Nets*. Springer, pp. 87–124. Available at `http://springerlink.metapress.com/openurl.asp?genre=article%&issn=0302-9743\&volume=3098\&spage=87`.

[6] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella & M. Massink (1998): *Automatic Verification of a Lip-Synchronisation Protocol Using Uppaal*. *Formal Aspects of Computing* 10(5-6), pp. 550–575. Available at `http://www.springerlink.com/content/21366fhg6r7xt15b/`.

[7] Conrado Daws & Piotr Kordy (2006): *Symbolic Robustness Analysis of Timed Automata*. In: *Formal Modeling and Analysis of Timed Systems, Lecture Notes in Computer Science* 4202. Springer Berlin / Heidelberg, pp. 143–155. Available at `http://www.springerlink.com/content/l8026102m7jv3j32/`.

[8] David L. Dill (1990): *Timing assumptions and verification of finite-state concurrent systems*. In: *Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science* 407. Springer Berlin / Heidelberg, pp. 197–212. Available at `http://www.springerlink.com/content/y053502404521143/`.

[9] Vineet Gupta, Thomas A Henzinger & Radha Jagadeesan (1997): *Robust Timed Automata*. In: *Hybrid and Real-Time Systems, LNCS* 1201. Springer-Verlag, pp. 331–345.

[10] T. A. Henzinger, X. Nicollin, J. Sifakis & S. Yovine (1994): *Symbolic Model Checking for Real-Time Systems*. *Information and Computation* 111(2), pp. 193 – 244. Available at `http://www.sciencedirect.com/science/article/B6WGK-45NJVYS-W/2/24e16e68e4c74a19199c87c5330433bf`.

[11] Thomas A. Henzinger & Jean-François Raskin (2000): *Robust Undecidability of Timed and Hybrid Systems*. In: *HSCC'00, LNCS* 1790. Springer-Verlag, pp. 145–159.

[12] Robin Milner (1995): *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

[13] Anuj Puri (1998): *Dynamical Properties of Timed Automata*. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 210–227. Available at `http://www.springerlink.com/content/c432527n7g271qk1/`.

[14] Anuj Puri (2000): *Dynamical Properties of Timed Automata*. *Discrete Event Dynamic Systems* 10(1-2), pp. 87–113. Available at `http://www.springerlink.com/content/n2q726540t250p41/`.

[15] Tim Regan (1993): *Multimedia in Temporal LOTOS: A Lip-Synchronization Algorithm*. In: *PSTV*. North-Holland Publishing Co., pp. 127–142.

[16] Jean-Bernard Stefani, Laurent Hazard & François Horn (1992): *Computational model for distributed multimedia applications based on a synchronous programming language*. *Computer Communications* 15(2), pp. 114–128. Available at `http://dx.doi.org/10.1016/0140-3664(92)90131-W`.

[17] Department of Information Technology at Uppsala University & the Department of Computer Science at Aalborg University (2008). *UPPAAL home page*. `http://www.uppaal.com/`.

[18] Howard Wong-Toi (1997): *Analysis of Slope-Parametric Rectangular Automata*. In: *Hybrid Systems V, Lecture Notes in Computer Science* 1567. Springer Berlin / Heidelberg, pp. 390–413. Available at `http://www.springerlink.com/content/gbpdqhw4k8vq1d0l/`.

[19] Martin De Wulf, Laurent Doyen, Nicolas Markey & Jean-François Raskin (2004): *Robustness and Implementability of Timed Automata*. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Lecture Notes in Computer Science* 3253. Springer Berlin / Heidelberg, pp. 118–133. Available at `http://www.springerlink.com/content/ud8lq155dwcm7mrf/`.

[20] Martin De Wulf, Laurent Doyen, Nicolas Markey & Jean-François Raskin (2008): *Robust safety of timed automata*. *Formal Methods in System Design* 33(1-3), pp. 45–84. Available at `http://www.springerlink.com/content/l7416573x425785m/`.

[21] Mihalis Yannakakis & David Lee (1993): *An efficient algorithm for minimizing real-time transition systems*. In: *Computer Aided Verification, Lecture Notes in Computer Science* 697. Springer Berlin / Heidelberg, pp. 210–224. Available at `http://www.springerlink.com/content/p72t4041h860j677/`.

Rational Physical Agent Reasoning Beyond Logic

Sandor M Veres and Nick K Lincoln
University of Southampton, Highfield, SO17 1BJ, UK, Email. s.m.veres@soton.ac.uk

Abstract

The paper addresses the problem of defining a theoretical physical agent framework that satisfies practical requirements of programmability by non-programmer engineers and at the same time permitting fast realtime operation of agents on digital computer networks. The objective of the new framework is to enable the satisfaction of performance requirements on autonomous vehicles and robots in space exploration, deep underwater exploration, defense reconnaissance, automated manufacturing and household automation.

1. Introduction

Agent theory has a substantial history within the field of computer science and has been theorized over greatly, stemming from the widely held view that an agent is a system most appropriately described by the intentional stance, wherein the agent is an entity subject to anthropomorphism [1]. Whilst, to date, there is no panacea for agent theory, significant contributions have been made relating to what properties an agent should have and how these properties should be formally represented and reasoned about [1,2,3,4,5]. Application areas for agent systems have been presented for the realms of software, industry and autonomous control systems [1,5,6].

Agent architectures are divisible into three groups: reactive, deliberative and hybrid. Reactive architectures operate through a mapping of data to action, in a stimulus-response fashion, as exhibited by Brook's subsumption architecture. Subsumption architectures present intelligence as an emergent property resultant from a hierarchy of finite state machines such that there is a behavioral response resultant from goal oriented desires; there is no form of planning. Such simplistic architectures are advantageous in dynamic environments, though in addition to the difficulty of engineering a set of behaviors to achieve a particular task, undesired behaviors may also evolve. Moreover, omitting the ability for such agents to reason eliminates the possibilities of planning and learning: both highly desirable proactive traits for agents [7,8].

Deliberative architectures can be separated into symbolic and intentional systems; these latter architectures are based upon abstracted notions written within the intentional stance. Intentional systems, as philosophized over by Dennett, encompass system descriptions abstracted out using human notions such as belief and intent [9]. Such abstractions allow irrelevant details to be omitted and hence save in computational complexity, permitting subsequent reasoning to be based upon high level notions and not modeling physics as would be resultant from modeling within the physical stance. Both deliberative theories are strongly bound to logical systems and it is these engines which are used for agent reasoning. The earliest cited use of such logical formalism within agent reasoning is that of the possible world semantics, as presented by Hintikka, now commonly formulated in a Kripke structure using normal modal logic [1]. Unlike reactive systems, deliberative architectures do not imply any form of output, merely theorizing over system knowledge. By definition an agent is proactive, thus requiring some form of output: early work involving the integration of action and knowledge is from Moore, who presents a formalism wherein knowledge is used as a pre-condition for action. This is in agreement with Hintikka, who postulated that not only is knowledge is the goal of inquiry, but what

'decisions and actions are based upon'. This work was closely followed by a theory of intention by Cohen and Levesque [10].

Symbolic architectures are logical systems, wherein the environment is represented symbolically and manipulated through reasoning. It is common practice now that the symbolic representations are logical formulae and it is these which are manipulated: this syntactic manipulation is theorem proving. Such manipulation permits an agent not only to reason about the consequence of current percepts, but states of the world which it may bring about. Although advantageous in both mathematical rigor and concept, since human knowledge can be considered as symbolic, it is difficult to translate the world effectively into symbolic representation and the computation times required for formal proofs generally precludes implementation on real time systems, even if using a concurrent theorem proving mechanism [10] .

Intentional systems are rooted in philosophy and based upon logical reasoning of the intentional stances used to represent the agent. Epistemic and doxastic logic, the logic of knowledge and belief respectively, are two forms of logic which fit within the anthropomorphist nature of an agent. A logical framework wherein beliefs, desires and intentions are primitive attitudes, as formulated by Rao and Georgeff, has proven to be the most popular: the three BDI primitives are highly cited within literature and used within numerous agent programming languages including the logic based PRS, Jason and 3APL, as well as the Java based implementations of JADE, Jadex and JACK [2,3,4].

Systems aiming to combine the timely nature of reactive architectures and the mathematical rigor of deliberative architectures are termed as hybrid, or layered architectures. Layered systems, as their name would suggest, involve a hierarchy of interacting subsystem layers; these layers may be horizontal or vertical. Complexities of information bottlenecks and the need for internal mediation within horizontal architectures are partially alleviated within vertical architectures, though these structures do not provide for fault tolerance. Both horizontal and vertical architectures have been implemented within TouringMachines and InterRRaP respectively [12].

Possibly the most widely known BDI implementation is PRS [13, 14]. PRS is a situated real time reasoning system that has been applied to the handling of space shuttle malfunctions, threat assessment and the control of an autonomous robot. Within the PRS framework a knowledge database, detailing how to achieve particular goals or react to certain situations, is interfaced by an interpreter. Conceptually this is similar to a horizontally layered TouringMachine, though notions if intention are contained and consequently placing PRS as a deliberative BDI model. Whilst certainly a notable and proven method, the knowledge database is fixed and invoked upon perceived condition triggers for the particular knowledge item of relevance. The set of possible agent behaviors is restricted by the number of knowledge items programmed and held within the library: an agent cannot add knowledge items during runtime, self generated or otherwise.

Here we propose interleaving of physical stances within the intentional stances used to abstract and reason about the system in order to enrich the reasoning process. Some might question the validity of such an approach as beneficial to aiding means–end reasoning, since conceptually we are providing an elaborate plan mechanism, which could be instigated within the PRS architecture. However, PRS interprets percepts and in conjunction with goals, will select a particular pre-formed plan to run: we are proposing the possibility for physical stances to act internal to deliberation as well as being the result of deliberation; not instigated as a result of deliberation until a preferential alternative is selected.

2. An agent framework for unstructured physical environments

A physical agent system (PAS) is a 3-tuple consisting of the agent, the environment in which the agent exists and a coupling between the agent and environment.

Definition A *physical agent system,* α, is defined by

$$\alpha = \langle A|E|C\rangle$$

where A is a set of agents, $A = \{A^i \big| i \in S\}$, E is a set of environment objects, $E = \{E^i \big| i \in S\}$, and C is a set of couplings between $\Gamma \subseteq \{\gamma\langle a, b\rangle | a, b \in A \cup E\}$ and any members of the joint set $A \cup E$. Couplings between members of A are called communications, $\boldsymbol{C}$, and couplings between members of E are called physical interactions, $\boldsymbol{P}$. Couplings between members of A and E are called action and sensing, Æ, in the environment.

A coupling is an abstract object at this stage of our discussion that will be specified later for its types, attribute and models (see Section 4). In this paper we are interested in physical agents that are not pure software agents but ones that have a set of environmental objects associated with them and are called the "body" of the agents. Strictly speaking a body is a set of environmental objects associated with a set of agents.

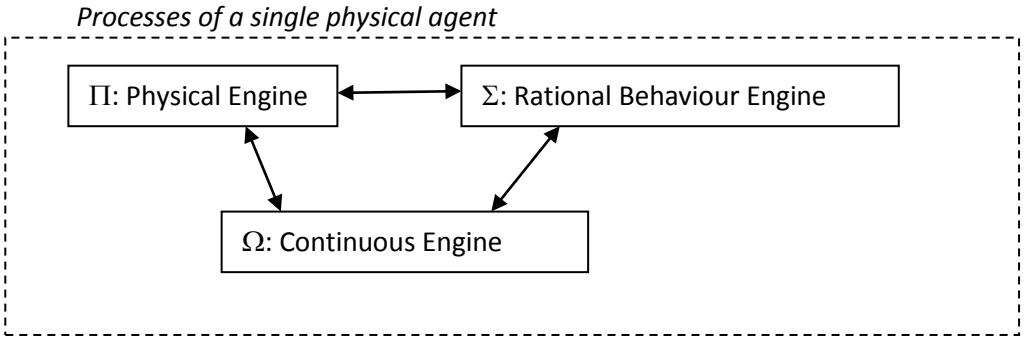
Definition The *body association* of agents is a map, *β,* defined by splitting A and E into a disjoint set of sets of agents and sets of environmental objects, respectively:

$A = \bigcup A_k$, $E = \bigcup E_k$,

and allocating to each A_k a subset of E : $A_k \overset{\beta}{\rightarrow} E_k$ where $E_k \subseteq E$.

In exceptional circumstances the $E = \bigcup E_k$ may be allowed not to be disjoint, meaning that agent teams can "share some body parts", but this is not the norm. For logical clarity one should organize teams of agents, such that $E = \bigcup E_k$ is a disjoint union.

Definition. A single *physical agent* is a tuple $A = \langle \Pi|\Sigma|\Omega\rangle$ consisting of the three main constituents of any agent: its physical engine, its rational behavior engine and its continuous engine.



This formal representation of a physical agent has been created to serve three practical purposes in engineering autonomous systems that interact with the physical environment:

1. *Easy programming* by engineers through defining sensing and actuation for the physical engine, definition of prediction in the continuous world (using the continuous engine) and by defining goals and behaviour constraints for the rational behavior engine.

2. *Formal verifiability* of the complete hybrid system of the agent system and its environment class, based on assumptions made on probabilities of physical engine malfunctions, numerical precision of the continuous engine and a hybrid system model-class of the environment.
3. *Fast realtime operation* on digital computers with asynchronous processes running in parallel. Any physical agent must have at least one process for each of Π, Σ, Ω . Realtime functionality may well require that some or all of Π, Σ, Ω are split into several, or many, sub processes.

If any of these practical requirements are not met then the uptake of the agent system by industry can be seriously limited.

The physical engine can be a complex process or set of processes. It can contain a multitude of devices, including communication, sensor and actuator devices, as well as sensor data abstractors (SDAs), control data developers (CDDs) and feedback processes (FBPs) for realtime processing of perception and actions of the agent. The SDAs are to symbolize events for Σ, the CDDs are to develop symbolic action into control signals for the actuators and the FBPs are needed for linking CDD and SDA pairs into realtime feedback loop executors.

Definition. A *physical engine* Π is a quad tuple, $\Pi = \langle \gamma | \sigma | \alpha | \Delta \rangle$, where γ is a set of communication devices, σ is a set of sensor data abstractors, α is a set of control data developers and Δ is a set of realtime feedback loop executors. For practical reasons any of σ, α, Δ may be empty, but this is not true for γ, which represents communication devices and consequently may never be empty.

Definition. A *software agent* is a physical agent that has only communication devices present within its physical engine, i.e. all of σ, α, Δ are empty.

The rational behavior engine, Σ, is formally defined so as to permit the accommodation of simple reactive subsumption based architectures as well as deliberative belief-desire-intention, i.e. BDI agent architectures. Whatever agent architecture is implemented, they must handle sensed event abstractions, agent action abstractions and decision making processes (DMPs).

From a theoretical point of view, it is the DMPs where agent architectures mostly differ in complexity. Given a set of $\gamma, \sigma, \alpha, \Delta$ abstractions, one can define simple behavior rules for a reactive subsumption architecture, or one can define a deliberative agent decision making process. The latter can use a belief set that can influence goal list prioritization and commitments to intentions that are executed, while monitoring beliefs and changing action if necessary. Of course a BDI architecture may involve more complex $\gamma, \sigma, \alpha, \Delta$ abstractions, however the overall architecture described above remains valid. As the next section will show, humans formulate their behavior rules in abstract temporal logic statements, so to satisfy the requirement of "ease of programming", it would be inappropriate to ignore temporal logic as a fundamental component of the DMP. As described later, and as one of the main contributions of this paper, there will be another equally important component of any capable advanced agent, that is via Ω, the continuous engine. The Ω moves beyond logic and facilitates the handling of unknown unstructured environments by the agent.

A question remains relating to the complexity that the DMP going to take on. It is inevitable that the DMP will be implemented using the abstractions of $\gamma, \sigma, \alpha, \Delta$ plus some memory and suitable data structures. These data structures may possibly be different resolution maps of the world in which the agent resides that may be uncertain in their fine detail. Similar to the way we (as humans) use maps to learn the abstract layout of streets or connections of metro lines, robots may only have a rough and highly abstract guide to the world. In the case of the metro line maps even the rough geometry of where the lines go in the city is neglected and the map is only useful to know how the stations are connected. These examples from human context point to the importance of a robot needing maps with different resolutions to make goal achievement and planning fast. In an unknown environment, for which there are no detailed maps available, the robot must implement SLAM (self localization and mapping) techniques.

Apart from maps there may also be a need for a *memory for abstract skills* (SAM) execution. Although the Physical Engine has $\gamma, \sigma, \alpha, \Delta$ for the actual execution of physical contact with the world, physical skills are to be controlled at an abstract level in feedback/feedforward. Speed of execution in some skills may require sending out a sequence of abstract commands in α for execution, without concurrent feedback but rather details relating to the end result of the abstract command sequence. For humans this happens in driving a car where we learn what action to take in a particular situation, such a requirement to swerve or perform an emergency stop, and feedback can only be registered with some delay; speed of feedback is limited. Abstracted physical skills command sequences, and their feedback mechanisms, are learnt and improved upon by actually practicing the activity, a testament to the phrase "practice makes perfect". The same should be true for robots as it is too laborious for programmers to develop physical skills of robots and the DMP system. Additionally, self-learning skills also infers learning how skills fit into the DMP system: integrity of the DMP operation is preserved. This integrity may become fragmented if programmers develop code for physical skills of agents and this highlights the importance of *learning skills execution (LSE) and skills abstract memory (SAM)* at an abstract level using discrete symbols $\gamma, \sigma, \alpha, \Delta$.

Most physical agents developed so far do not have a DMP with LSE and SAM. Existing robots tend to link abstractions directly with a limited set of physical actions. Using LSE and SAM the set of performable actions can be several magnitude greater than the set of abstractions for sensing and actions in $\gamma, \sigma, \alpha, \Delta$, which may be called primitives of sensing and actions from which more complex ones can be built. Good choice of primitives in $\gamma, \sigma, \alpha, \Delta$ will fundamentally influence the sophistication of physical abilities that the robot will be able to learn via practice.

Other components of DMP can be:

- Planning of movements in the physical world
- High level planning of goal achievement using various resolutions of world maps (self-learned or acquired)
- Ultimate goals of operation and behaviour constraints at higher levels of abstraction

The following definition grasps the minimum of what the DMP must contain for a capable physical agent, without specifying the actual mechanisms of operation, i.e. no "agent architecture" is specified.

Definition. A rational behavior engine Σ is a tuple, $\Sigma = \langle W | M_p | M_g | C | G \rangle$, that contains a granulated multi-resolution and physical multi-domain world models *W,* abstract physical skills memory, M_p, goal achievement memory (problem solving memory), M_g , abstract formulation of behavior constraints, *C*, abstract formulation of short and long term goals, *G*.

Although no specific agent architecture is required, we will briefly outline how some of the best known architectures fit into this. In all cases the $\gamma, \sigma, \alpha, \Delta$ are assumed to be common.

Subsumption architectures:

The Σ of a physical agent with "subsumption architecture" can for instance be filled in by the following:

W, M_p, M_g, G – Empty sets.

C – A set of "if A then B" rules where the premises A are propositional logic formulae in terms of abstraction symbols in γ or in σ. The conclusions are propositional logic formulae of symbols in .

BDI architectures in Jason or similar agent oriented languages:

W – Modeling structures in various physical domains and varying resolution

M_p, M_g – Plans and logic rules programmed in the agent programming language

G – Goal symbols that appear at the head of some plans

METATEM deliberative agents using temporal logic :

W – Modeling structures in various physical domains and varying resolution

M_p, M_g – Databases of symbol vectors in terms of $\gamma, \sigma, \alpha, \Delta$ and temporal propositional logic connections

G – Temporal logic statements in terms of higher level abstractions as explained by temporal logic formulas using abstraction from: $\gamma, \sigma, \alpha, \Delta$.

The latter realization is the most natural for humans in terms of goal definitions and executions of tasks in a logical manner. The question of modeling structures relating to the continuous world, and their role in planning and decision making, has so far been neglected. It is a fact however, that decisions can be influenced by hypothetical planning and not decision to be made first and then planning put into motion. This latter can lead to the agent making bad decisions. This fact and that initial abstractions in σ, as prescribed by the designer of the agent, may not be sufficient to capture the essence of the changing world correctly, leads us to the next section defining the "Continuous engine" of the physical agent.

Definition. The continuous engine, Ω, is a tuple $\Omega = \langle M|S|O|B|L \rangle$ where *M* is a set of approximate continuous models of the world, S is a continuous time simulator that uses analytical and empirical data based dynamic models to predict future state of the world, *O* is an optimizer that can optimize continuous time planning of actions, *B* is a Boolean evaluator of propositions in terms of σ statements and *L* is a library of useful numerical computations in terms of continuous variables.

As W is the primary, symbolic model of the world with relationships stored on symbols, the M is related to W. The M is a collection of models from various physical domains (geometric, mechanical, electrical, gravitational, heat, fluid flow, etc…) that make symbolic descriptions in W either more precise in numerical or in qualitative, time evolution sense. Even without precise numerics, the agent may perform a simulation using the simulation tools in S to see possible qualitative outcomes in terms of σ, α, Δ abstractions. These qualitative outcomes of predictions can be used to make the right decisions and planning by Σ.

The O is a set of optimization tools of continuous problems. The agent can perform planning in continuous time and can set up a problem formulation. Then it searches in O for a suitable optimization tool. The results are formulated and used at the symbolic level of σ, α, Δ. Optimization for path planning, robust feedback control or a low complexity numerical dynamical model may all be tools that are available in O.

B is important since Boolean values of primitives in σ in prediction (simulation) outcomes must be inferred by some continuous computations that cannot be performed elsewhere except in the continuous engine Ω. B contains a Boolean valued functional, by example "will the agent body be within the allowed boundary if it carries on moving the same way for the next 10s" is an evaluation that is not done in symbolic computation but using M, S and B: the statement is converted into a symbolic temporal logic statement for Σ.

Finally L is a library of auxiliary computations with continuous quantities: for instance equation solvers for linear systems of equations, nonlinear equation solvers and generic nonlinear optimizers. Use of L by the agent assumes that the agent is capable of setting up problems in Σ by first abstracting them and picking a solution tool from L.

It is evident that Σ and Ω run hand-in-hand, i.e. the Σ frequently delegates computations to Ω, and then uses the output from Ω to continue deliberation of prescribe action.

3. Programming of Physical Agents by Non-programmers

A human being is represented here as a "super agent" that has considerably more in its DMP then the formal agents to be engineered. The extra features of the human DMP can be briefly summarized as follows:

- Conceptual structures formed from $\gamma, \sigma, \alpha, \Delta$ abstractions.
- Seamless manipulation of conceptual structures to achieve lowest complexity models of the environment for goal achievement.
- Rich set of skills to cope with in various situations in a physical environment. This includes manual skills as well as skills of analytical modelling performed in the head or on a computer.
- Near "completeness" of abstract knowledge in an artificially created physical environment that permits reliable operation of human agents by design of the environment.

The latter is an interesting point: our educational system complements the human built environment to be safe for humans by providing skills to cope and introducing laws and unwritten conventions to render the human – physical world interactions relatively safe for mature adults.

Unfortunately, physical agents that we discuss in this paper do not enjoy the kind of advantages as listed. Instead the problem is to find a mapping of these human capabilities into formal agent properties that make them safe and achieving goals despite the very limited resources they have relative to humans. To do that we will formalize the human capabilities a bit further.

4. An Example Software Implementation Using J2M

There are many ways to implement the agent architecture described above. Ideally C, C++ could be used for all the components to permit operation on all platforms (Windows, Linux, Unix, MacOS and embedded systems) but there are both practical and legacy issues that make realization biased towards software systems that have ready tools available and which would be far too expensive to reproduce. For instance the MATLAB family of toolboxes provides a rich set of numerical processing, optimization and simulation algorithms that can be exploited. Similarly the Java environment has numerous software components in the area of agent reasoning that may be implemented directly. The following table provides some possible options for the software platform to be used for the implementation of the agent architecture outlined in Sections 2-4.

Π platform	**Σ platform**	**Ω platform**
C	C	C
C++	C++	C++
Java	Java	Java
MATLAB	Java	MATLAB
SIMPOL	SIMPOL	SIMPOL

In the following we describe a possible software implementation in terms of MATLAB+Java+MATLAB (J2M) that has the advantage of minimal programming effort due to the rich set of programming tools presently available. Within the J2M implementation, the Π and Ω platforms are developed within MATLAB, whilst the Σ

platform is exclusively developed using Java. Although it is theoretically possible to form a rational engine within a MATLAB framework, since MATLAB does not allow for multi-threaded code execution, this would be a restricting factor. Also, to do so would neglect the existing frameworks which have been developed in more suitable languages. It should be noted that the high level prototyping in J2M may be converted into concise machine code for embedded applications. A description of the J2M will follow, within which we are assuming the implementation of a physical agent, that is one for which σ, α, Δ are not empty.

The physical engine, Π, is an element within an environment; this environment may be purely numerical, instantiated within a virtual world or a true dynamic real world environment. Regardless of construct, the physical engine is capable of extracting relevant abstract data from the inhabited world to be encapsulated within σ and communicated to the rational engine, Σ, via γ. This is the only interaction mechanism Π has with Σ.

Within the developed implementation, the physical engine continually passes sensory information to the rational engine, which is used to update W, though the rational engine may chose to ignore information it receives.

Sensor data, σ, may relate to position and velocity information given in some coordinate frame, as would be required for roving vehicles such as UAVs or AUVs. σ may also relate to other functional data types such as temperature and pressure information, extracted from relevant sensors. The way in which σ is communicated to Σ is critical, since Σ must be in the position to update W correctly based upon information received via γ. FIPA provides a comprehensive treatment of software standards and specifications for interacting agents and agent based systems, to enable intelligible communication between agents and agent processes. For the instance of sensor data flow from the physical engine to the rational engine, communication is unidirectional and so the communication considerations are reduced to the rational agent requiring knowledge only of the communicating entity (in this case Π) and the subject matter (σ). W is updated as a consequence of σ, from which Σ will evaluate the appropriate actions and this may instantiate invocations of Ω.

During the rational agent deliberation cycle, a point may be reached wherein calls to the continuous engine are required. Instances of these calls have already been entered upon; here we shall concentrate on the data flow. The continuous engine is constructed within MATLAB and consequently all functionality of this system is achieved through use of the appropriate m-files. Σ requests execution of a component from Ω, either S, O, B or L, dependent upon the solution sought by the deliberation cycle occurring within Σ, and also requests a response in the form of a pointer, character array, numerical or Boolean value.

Communication between Σ and Ω is more complex than that of the data pushing performed by Π, since actions executed by Ω are conditional upon input from Σ, and in turn Σ expects some form of result. Consequently we are presented with the need for more elaborate communications protocols to ensure efficient interaction between Σ and Ω. Messaging from Σ is of the form $<m_c,c_c,r_c>$, whereby m_c is a character string specifying the particular m-file to be executed, based upon the data encapsulated within c_c and expecting the number of returned evaluations to be the number specified within r_c. Since Σ is aware of the continuous engine invocation instance, the data type being returned by Ω need not be specified. Return dialog from Ω to Σ is in a similar format, of the form $<m_r,r_r>$, wherein m_r signifies the m-file which was executed and r_r the result of the routine. This format could be interpreted within the FIPA framework as <in-reply-to, data>.

MATLAB is unable to function in a multi-threaded nature and so Ω is restricted to parallel processes only; this is not true for Σ, which is intrinsically multi-threaded: whilst waiting for a response from Ω, Σ is able to continue other forms of deliberation which are unrelated to problems associated with the current Ω task. An example of such an instance is that of vehicle path optimization: Σ requests execution of O from Ω based upon current σ and M. Whilst O is executing within Ω, Σ is still capable of monitoring σ from Π and dealing with any instances which may arise, such as the need for control reconfiguration. Upon completion of O within Ω, the

relevant response (in this instance a pointer to the file detailing the optimized path) is sent to Σ, from which the invoking thread may be resumed. Ultimately all threads within Σ result in some form of action to be completed by Π and this requires some form of assertion by Σ; note that the prescription of 'no-action' represents the simplest assertion which may be prescribed by Σ. In the same way that Π acts to 'push' data to Σ, Σ prescribes action for Π: here Σ prescribes the initiation of Δ which acts directly upon α. Returning to the example of path following, Σ initiates Δ within Π relating to the task of path following using the plan formed by Ω and accessed by the file pointer provided to Σ by Ω, which is consequently available to Δ. Δ executes without additional action from Σ, using internal feedback devices, though Σ is capable of over-riding the execution of a Δ.

All engines run concurrently: Π is continuously feeding data to Σ, which prescribes action to Π via γ and invokes intermittent communication with Ω. Whilst Ω and Π do not communicate directly, shared memory between these engines allows access of the results from Ω by Π: such an instance is invoked when Ω is tasked with path optimization. Here Ω is invoked by Σ to optimize a path, or thruster sequence, and this plan is stored within a file. A pointer to this file passed to Σ, which communicates this pointer to Π. If instructed to follow the plan, Π reads the file directly and acts accordingly. A schematic of the J2M construct, indicating interaction of the three agent engines, is given within Figure 1.

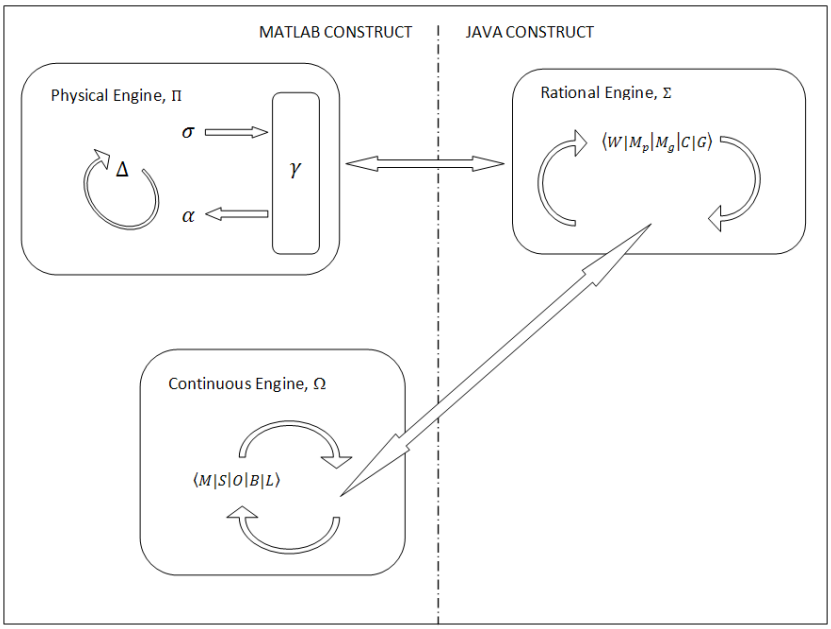


Figure 1: J2M Framework Construct

An example implementation of the presented agent framework has been developed based upon a spacecraft agent scenario. Within the scenario a spacecraft agent is tasked with acquiring and tracking a geostationary orbital location upon a failed orbital insertion. During the simulation the agent is also presented with instances of actuator failure, requiring control reconfiguration in order to achieve the mission objective.

The agent world is developed within Simulink (MATLAB), and is inclusive of disturbances resulting from Earth oblateness (triaxiality), solar radiation pressure and Luni-Solar third-body perturbations [15]. The agent is not aware of these impacting factors: within the continuous engine, Ω, the relevant physics engine for internal simulation which may be utilized are based upon the Hill equations, a simplification of the orbital dynamics with respect to the desired orbital location [16]. The spacecraft agent is availed with numerous possible control methodologies, afforded by the considerable amount of literature available on the subject of

spacecraft control, inclusive of planning, adaptive and reactive control systems [17-23]. It is the task of the reasoning engine to select an appropriate control methodology to deal with the presented scenario, perform the appropriate control and analyse the resultant output in order to improve performance. Concurrent to the control requirements, the spacecraft agent is tasked with monitoring internal systems and reacting accordingly to any diagnostics made. Within simulation, the actuators present the agent with situations of gain reduction and propellant supply issues resulting from supply valves being stuck open/closed and a fuel-line breach.

The upper level Simulink block is shown within Figure 2, with the VRML sink linking to a virtual world shown in Figure 3. Whilst not essential, it was chosen to include a VRML world to aid visualization of the dynamical processes occurring as a result of agent action or inaction. The world is initialized with the agent in an undesired location and with a state representative of a failed orbit insertion. A sphere indicates the desired geostationary location bounds: when within these bounds the sphere surface is green; when in violation of the bounds the surface is red. Note that for visualization purposes, scales have been altered greatly.

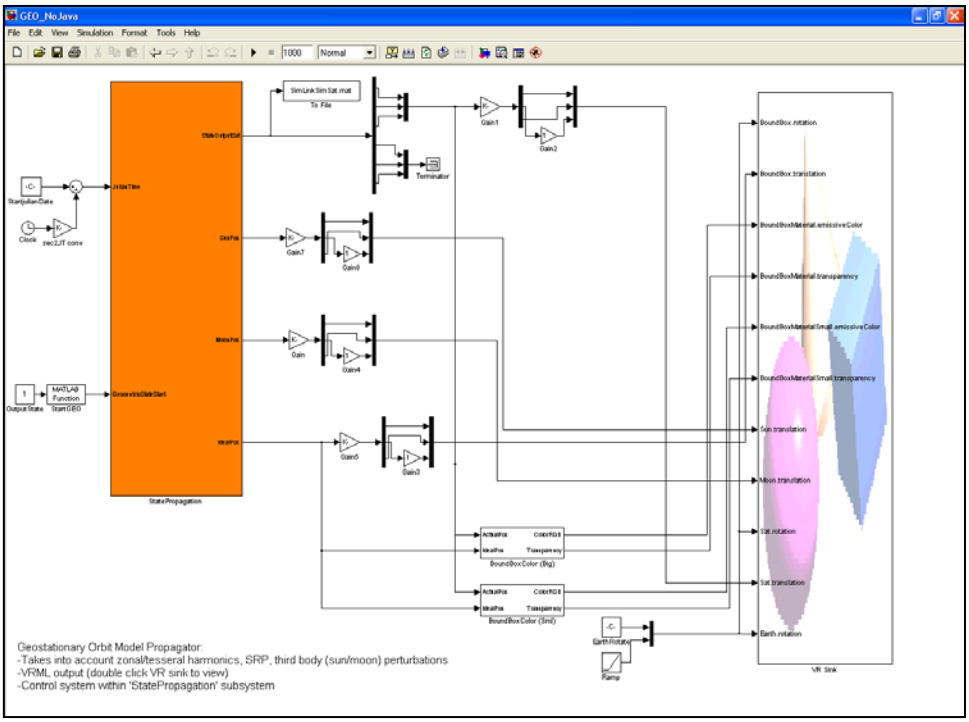


Figure 2: Agent Simulink Environment

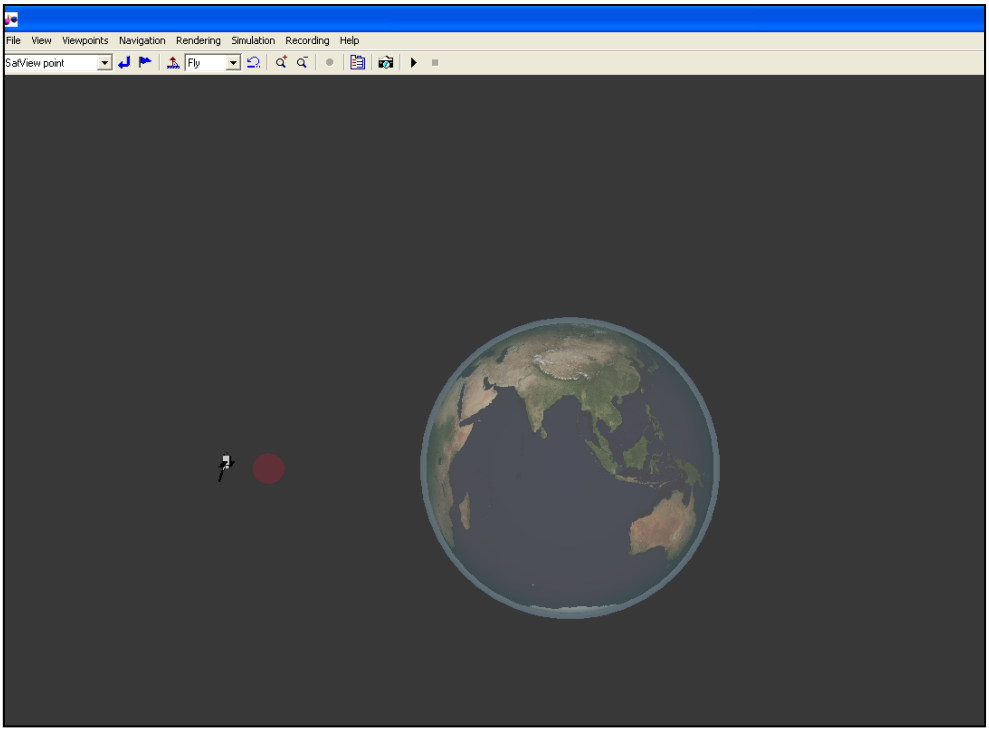


Figure 3: Agent VRML World Initialized State

Agent percepts are taken from the simulated world, from which abstractions are made and passed to the rational engine developed within Java. As presented within the proposed agent framework, the rational engine may task the continuous engine and these results may be used within deliberation to result in a rationalized prescription of action: agent action is currently limited to thruster actuation and internal reconfiguration to adapt for faulty thrusters.

Using the presented framework within the simulation, successful geostationary orbit attainment and regulation has been achieved in the presence of unknown disturbances and actuator failure, necessitating online control reconfiguration.

Summary

This paper has discussed and presented a theoretical physical agent framework that satisfies the practical requirements of programmability by non-programmer engineers and realtime operation of deployed agents. The presented framework has been developed and implemented within simulation for autonomous spacecraft control tasked with position tracking, subject to orbit insertion error, unmodeled orbital perturbations and control reconfiguration requirements under the presence of actuator performance degradation and failure. Further development of this model, and the extension of the existing framework to additional agent scenarios, is the subject of continued research.

References

[1] Michael Wooldridge and Nicholas R. Jennings, *Intelligent Agents: Theory and Practice*. Knowledge Engineering Review Journal, 1995, volume 10, pp 115-152.

[2] Anand S. Rao and Micheal P. Georgeff, *Formal Models and Decision Procedures for Multi-Agent Systems*. Technical note 61: Australian Artificial Intelligence Institute, June 1995.

[3] Rafael H. Bordini, Jomi F. Hubner and Micheal Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Published by John Wiley and Sons, 2007

[4] Fabio Bellifemine, Giovanni Caire and Domonic Greenwood, *Developing Multi-Agent Systems With Jade*. Published by John Wiley and Sons, 2007.

[5] H. Van Dyke Parunak, *Practical And Industrial Applications of Agent-Based Systems,* 1998.

[6] Sandor M. Veres, *Autonomous Control Systems Using Agents: An Introduction*. In, Proceedings of IEE Workshop on Agent Based Control Systems. IET (IEE), 1-10 (2005).

[7] Sandor M. Veres and Aron G Veres, *Learning and Adaptation in Physical Agents.* In, *9th IFAC Workshop "Adaptation and Learning in Control and Signal Processing" (ALCOSP'07), St Petersburg, Russia, 29-31 Aug 2007.* , 6pp.

[8] Sandor M. Veres and Aron G. Veres, *Learning and Adaptation of Skills in Autonomous Physical Agents.* In, *17th World Congress of International Federation of Automatic Control (IFAC), Seoul, Korea, 6-10 Jul 2008*. Seoul, Korea, Elsevier - Pergamon Press, 6pp, 2671-2676.

[9] Daniel C. Dennett, *The Intentional Stance*. Published by MIT Press, 1989.

[10] Cohen, P. R. and Levesque, H. J. (1990). *Intention is choice with commitment*. *Artificial Intelligence*, 42:213-26

[11] Michael Fisher, *An Open Approach to Concurrent Theorem-Proving*. Technical report, Department of Computing, Manchester Metropolitan University.

[12] Michael Wooldridge, *An Introduction to MultiAgent Systems*. Published by John Wiley and Sons, 2002.

[13] Georgeff, M. P. and Lansky, A. L. (1987). Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677-682, Seattle, WA.

[14] K. L. Myers, *Procedural Reasoning System User's Guide: A Manual for Version 2.0*. Technical report, Artificial Intelligence Center, SRI International, Menlo Park, CA, 2001.

[15] David A. Vallado and Wagne D. McClain, *Fundamentals of Astrodynamics and Applications (Space Technology Library)*. Published by Springer Link, 2001.

[16] Marcel J. Sidi, *Spacecraft Dynamics and Control*, Published by Cambridge University Press,1997.

[17] S. M. Veres, S. B Gabriel, D. Q Mayne and E. Rogers, *Analysis of Formation Flying Control of a Pair of Nano-satellites*. AIAA Journal of Guidance, Control, and Dynamics, 25 (5). pp. 971-974.

[18] R. Pongvthithum, S. M. Veres, S. B. Gabriel and E. Rogers, *Universal Adaptive Control of Satellite Formation Flying*, International Journal of Control, Volume 78(1), January 2005

[19] D. Ya. Rokityanski and S.M. Veres, Application of Ellipsoidal Estimation to Satellite Control. *Mathematical and Computer Modelling of Dynamical Systems*, 11, (2), pages 239-249 (2005).

[20] N.K. Lincoln and S.M. Veres. *Components of a Vision Assisted Constrained Autonomous Satellite Formation Flying Control System*. International Journal of Adaptive Control and Signal Processing, 21(2-3):237–264, October 2006.

[21] S.M. Veres, (2006) Autonomous formation flying of satellite robots: the mechanical control layer. In Proceedings, *TAROS 2006: Towards Autonomous Robotic Systems, Guildford, UK, 4-6 Sep 2006*. .

[22] S.M. Veres, Thanapalan, K., Gabriel, S. and Rogers, E. (2006) Reconfigurable Controller Design For Operational Safety in Satellite Formation Flying. In Proceedings, *International Control Conference 2006, Glasgow, Scotland, UK, 30 Aug - 1 Sept, 2006*.

[23] Sandor M. Veres and Nicholas K. Lincoln, Sliding Mode control for Agents and Humans. In Proceedings, *TAROS'08, Towards Autonomous Robotic Systems 2008, Edinburgh, UK, 1-3 Sept 2008*. Edinburgh, UK, IC London, 13pp, 61-73.

Observability and Diagnosability of Hybrid Systems with Application in Air Traffic Management

Maria Domenica Di Benedetto[1], Stefano Di Gennaro[2], and Alessandro D'Innocenzo[3]

DEWS Centre of Excellence for Research,
Department of Electrical and Information Engineering, University of L'Aquila, Italy.

1. dibenede@parades.rm.cnr.it
2. digennar@dis.uniroma1.it
3. adinnoce@seas.upenn.edu

October 14, 2009

Abstract

In an Air Traffic Management (ATM) closed-loop system with mixed computer-controlled and human-controlled subsystems, recovery from non-nominal situations implies the existence of an outer control loop which has to identify these situations and act accordingly to prevent them to evolve into accidents. The purpose is to develop algorithms with guaranteed performances for assisting human operators in detecting critical situation and avoiding the propagation of errors and other non–nominal events. Estimation methods and observer design techniques are essential in this regard for the design of a control strategy for error propagation avoidance and/or error recovery. Various aspects need to be taken into account in the study of error detection for ATM: psychological models which can be used for the study of ATM; stochastic hybrid models describing the dynamics involved in error evolution control, capturing the essential features of ATM; observability/diagnosability and observer/diagnoser design for these hybrid models; the applicability of theoretical results on observers to a realistic ATM situation.

We propose a novel definition of observability motivated by safety critical applications given with respect to a subset of *critical* discrete states, that model unsafe or unallowed behaviors. For the class of discrete event systems, we address the problem in the setting of formal (regular) languages and propose a novel observability verification algorithm. For the class of switching systems, we characterize the minimal set of extra output information to be provided by the continuous signals in order to satisfy observability conditions, and propose a milder observability notion that allows a bounded delay in state observation. For the class of hidden Markov models, we analyze decidability and complexity of the verification problem.

We also introduce a definition of diagnosability, which corresponds to failure detection in finite time. Given a plant, a system is diagnosable if, within a finite time bound and only using the observable output of the plant, it is possible to detect that a fault has occurred. The concept of diagnosability is tightly related to observability: diagnosability generalizes observability. We tackle diagnosability verification for hybrid automata using an abstraction technique. We propose a procedure for constructing an abstraction of a hybrid automaton, which belongs to a subclass of timed automata called durational graph. We prove, as a second contribution of this paper, that the proposed diagnosability verification problem belongs to the complexity class P.

Formalization and Validation of Safety-Critical Requirements*

Alessandro Cimatti
FBK-irst
Trento, Italy
cimatti@fbk.eu

Marco Roveri
FBK-irst
Trento, Italy
roveri@fbk.eu

Angelo Susi
FBK-irst
Trento, Italy
susi@fbk.eu

Stefano Tonetta
FBK-irst
Trento, Italy
tonettas@fbk.eu

The validation of requirements is a fundamental step in the development process of safety-critical systems. In safety critical applications such as aerospace, avionics and railways, the use of formal methods is of paramount importance both for requirements and for design validation. Nevertheless, while for the verification of the design, many formal techniques have been conceived and applied, the research on formal methods for requirements validation is not yet mature. The main obstacles are that, on the one hand, the correctness of requirements is not formally defined; on the other hand that the formalization and the validation of the requirements usually demands a strong involvement of domain experts.

We report on a methodology and a series of techniques that we developed for the formalization and validation of high-level requirements for safety-critical applications. The main ingredients are a very expressive formal language and automatic satisfiability procedures. The language combines first-order, temporal, and hybrid logic. The satisfiability procedures are based on model checking and satisfiability modulo theory. We applied this technology within an industrial project to the validation of railways requirements.

1 Introduction

Formal methods are widely used in the development process of safety-critical systems. The application of formal verification techniques relies on the formalization of the system's design into a mathematical language. Several formal languages are available according to the different aspects that are relevant to the verification, and many design tools can automatically formalize the design into one of these languages.

Hybrid systems are often used to formalize safety-critical designs because they can capture the interaction between an embedded system and its environment. In particular, hybrid automata combine a finite state machine representing the discrete controller with constraints over dense-domain variables that represent the physical entities.

Many techniques have been conceived in order to verify the design according to the required level of automation and the expressiveness of the property of interest. State-of-the-art approaches mix *model checking* [CGP99] and *theorem proving* in order to tackle the verification of infinite-state systems with a sufficient level of automation.

Another important aspect of the development process is the correctness of the *requirements*. Very often bugs in the late phases are caused by some flaws in requirements specification. These are difficult to detect and have a huge impact on the cost of fixing the bug. Nevertheless, formal methods on requirements validation are not yet mature. In particular there is no precise definition of correct requirements.

The most relevant solution has been proposed in the context of the *property-based approach* to design, where the development process starts from listing a set of formal properties, rather than defining

*A. Cimatti, M. Roveri, and A. Susi have been partly supported by the European Railway Agency under the project EuRailCheck, service contract ERA/2007/ERTMS/02. S. Tonetta has been supported by the Provincia Autonoma di Trento (project ANACONDA).

Submitted to:
FMA 2009

an abstract-level model. The requirements validation is performed with a series of checks that improve the confidence in the correctness of the requirements. These checks consist of verifying that the requirements do not contain contradictions, that they are not too strict to forbid desired behaviors, that they are not too weak to allow undesired behaviors. This process relies on the availability of a sufficiently expressive logic so that properties as well as desired and undesired behaviors can be formalized into formulas. The approach considers a one-to-one mapping between the properties and the logical formulas. This allows for traceability of the formalization and the validation results, and for incremental and modular approaches to the validation.

In the context of *safety-critical applications*, the choice of the language used to formalize the requirements is still an open issue, requiring a delicate balance between expressiveness, decidability, and complexity of inference. The difficulty in finding a suitable trade-off lies in the fact that the requirements for many real-world applications involve several dimensions. On the one side, the objects having an active role in the target application may have complex structure and mutual relationships, whose modeling may require the use of rich data types. On the other side, static constraints over their attributes must be complemented with constraints on their temporal evolution.

One of the main obstacle in applying this approach to the industrial level is that requirements are often written in a natural language so that a *domain knowledge* is necessary both to formalize them and to define which behaviors are desirable and which not during the validation process. Since domain experts are typically not advanced users of formal methods, we should provide with a rich but friendly language for the formal specification and an automatic but scalable engine for the formal verification.

In this paper, we report on a methodology and a series of techniques that we developed for the formalization and validation of high-level requirements for safety-critical applications. The methodology is based on a three-phases approach that goes from the informal analysis of the requirements, to their formalization and validation [CRST08a]. The methodology relies on two main ingredients: a very expressive formal language and automatic satisfiability procedures. The language combines first-order, temporal, and hybrid logic [CRST08b, CRST09, CRT09]. The satisfiability procedures are based on model checking and satisfiability modulo theory. We applied this technology within an industrial project to the validation of railways requirements. The tool [CCM+09] integrates, within a commercial environment, techniques for requirements management, for model-based design, and advanced techniques for formal validation with the model checker NuSMV [CCGR00].

The rest of the paper is organized as follow: in Section 2, we outline the proposed methodology, giving details on the chosen language in Section 2.1 and on the validation procedure in Section 2.2; in Section 3, we describe the project where the methodology was applied; in Section 4, we review the related work, and in Section 5, we conclude.

2 A methodology for the formalization and validation of requirements

Our methodology has been presented in [CRST08a]. It consists of three main steps:

- *Informal analysis.* The first activity in the methodology is the informal analysis of the set of requirements. In this phase, first the requirement fragments are identified and categorized on the basis of their characteristics. Then, they are structured according to their dependencies.
- *Formalization.* The second phase consists of the formalization of each categorized requirement fragment identified in the informal analysis by specifying the corresponding formal counterpart. The link between informal and formal is used for requirements traceability of the formalization

against the informal textual requirements, and to select directly from the textual requirements document a categorized requirement fragment to validate.

- *Formal validation.* The third phase aims at improving the quality of the requirements. This goal is achieved by performing several analysis, based on the use of formal techniques, which may help to pinpoint flaws that are not trivial to detect in an informal setting. It consists of the definition of a series of validation problems and the analysis of the results given by an automatic validation check. These problems include checks to identify inconsistencies, and to increase the confidence that the categorized requirement fragment and its corresponding formalized counterpart meet the design intent: for instance, a flaw may be in the fact that some desired behaviors have been ruled out by an over-constraining set of requirements; conversely, some undesired behavior may have not been ruled out by under-constraining requirements.

There are three main checks that are performed to validate the selected set of formalized requirement fragments, namely, checking logical consistency, scenario compatibility, and property entailment:

- *Logical Consistency* to formally verify the absence of logical contradictions in the considered formalized requirement fragments. It is indeed possible that two formalized requirement fragments mandate mutually incompatible behaviors. Note that this check does not require any domain knowledge.
- *Scenario compatibility* to verify whether a scenario is admitted given the constraints imposed by the considered formalized requirement fragments. Intuitively, the check for scenario compatibility can be seen as a form of simulation guided by a set of constraints. The behaviors used in this phase can be partial, in order to describe a wide class of compatible behaviors. The check for scenario compatibility can be reduced to the problem of checking the consistency of the set of considered formalized requirement fragments with the constraint describing the scenario.
- *Property entailment* to verify whether an expected property is implied by the considered formalized requirement fragments. This check is similar in spirit to model checking, where a property is checked against a model. Here the considered set of formalized requirement fragment plays the role of the model against which the property must be verified. Property checking can be reduced to the problem of checking the consistency of the considered formalized requirement fragments with the negation of the property.

If one of the check reveals a problem, two causes are possible: the first one is that the formalization is not correct due to an improper use of the formal language or to an ambiguity of the informal specification; the second possibility is that there is a flaw in the informal specification that need to be corrected. An inspection of the diagnostic information can be carried out in order to discriminate among the two possibilities in order to take the most appropriate corrective action.

In fact, the above checks not only produce a yes/no answer, but they can also provide the domain expert with diagnostic information of different forms:

- *Traces*. When consistency and scenario checking succeeds, it is possible to produce a trace witnessing the consistency, i.e. satisfying all the constraints in the considered formalized requirement fragments. Similarly, when a property check fails the tool provides a trace witnessing the violation of the property by the formalized requirement fragments.
- *Unsatisfiable core*. If the specification is inconsistent or the scenario is incompatible, no behavior can be associated to the considered formalized requirement fragments; in these cases, the tool can also generate diagnostic information in the form of a minimal inconsistent subset. This information can be given to the domain expert, to support the identification and the fix of the flaw.

2.1 A property specification language for safety-critical applications

The success of the methodology relies on the availability of a specification language which is enough expressive to represent the requirements of safety-critical applications, and enough simple to be used by domain experts and analyzed with automatic techniques.

In order to specify requirements in the context of safety-critical applications we adopt a fragment of *first-order temporal logic*. The first-order component allows to specify constraints on objects, their relationships, and their attributes, which typically have rich data types. The temporal component allows to specify constraints on the temporal evolution of the possible configurations. We enriched the logic with constructs able to specify hybrid aspects of the objects' attributes such as derivatives of the continuous variables and instantaneous changes of the discrete variables. The logical formulas are consequently interpreted over hybrid traces where continuous evolutions alternate with discrete changes. Finally, the logic has been designed in order to be suitable for an automatic analysis with model checking techniques.

As described in [CRST09], we use a class diagram to define the classes of objects specified by the requirements, their relationships and their attributes. The class diagram basically defines the signature of the first-order temporal logic. The functional symbols that represent the attributes and the relationships of the objects are flexible in the sense that their interpretation change at different time points. Quantifiers are allowed to range over the objects of a class, and can be intermixed with the temporal operators.

The basic atoms of the logic are arithmetic predicates of the attributes and relationships of objects. As described in [CRT09], the "next" operator can be used to refer to the value of a variable after a discrete change, while the "der" operator can be used to refer to the first derivative of continuous variables during a continuous evolution.

The temporal structure of the logic encompasses the classical linear-time temporal operators combined with regular expressions. This combination is well established in the context of digital circuits and forms the core of standard languages such as the Property Specification Language (PSL) [EF06].

On the lines of PSL, we also provide a number of syntactic sugar which increases the usability of the language by the domain experts. This includes natural language expressions that substitute the temporal operators, the quantifiers, and most of the mathematical symbols. The resulting language resembles the Controlled Natural Language (CNL) used in [NF96].

2.2 Model checking techniques for requirements validation

The validation process of the proposed methodology relies on a series of satisfiability checks: consistency checking is performed by solving the satisfiability problem of the conjunction of the formalized requirements. The check that the requirements are not too strict is performed by checking whether the conjunction of the requirements and the scenario's formulas is satisfiable. Finally, the check that the requirements are not too weak is performed by checking whether the conjunction of the requirements and the negation of the property is unsatisfiable.

Unfortunately, the satisfiability problem of the chosen language is undecidable. The undecidability comes independently from the combination of temporal and first-order logics, from the combination of the uninterpreted functions and quantifiers, and from the hybrid component of the logic.

Nevertheless, we want to keep such expressiveness in order to faithfully represent the informal requirements in the formal language. Thus, we rely on automatic albeit incomplete satisfiability procedures.

First, we fix a number of objects per class so that it is possible to reduce the formula to equi-satisfiable one free of quantifiers and functional symbols [CRST09]. As described in [CRST08b], we can automat-

ically find a bound on the number of objects for classes under certain restrictions.

Second, we translate the resulting quantifier-free hybrid formula into an equi-satisfiable formula in the classical temporal logic over discrete traces. In this case, we exploit the linearity of the constraints over the derivatives to guarantee the existence of a piecewise-linear solution and to encode the continuity of the continuous variables into quantifier-free constraints.

Third, we compile the resulting formula into a Fair Transition System (FTS) [MP92], whose accepted language is not empty iff the formula is satisfiable. For the compilation we rely on the works described in [CRT08, CRST08b]. We apply infinite-state model checking techniques to verify the language emptiness of the resulting fair transition system. In particular, we used Bounded Model Checking (BMC) [BCCZ99], particularly effective in solving the satisfiable cases and producing short models, and Counterexample-Guided Abstraction Refinement (CEGAR) [CGJ$^+$00], more oriented to prove the unsatisfiability cases.

The language non-emptiness check for the FTS is performed by looking for a lasso-shape trace of length up to a given bound. We encode this trace into an SMT formula using a standard BMC encoding and we submit it to a suitable SMT solver. This procedure is incomplete from two point of views: first, we are performing BMC limiting the number of different transitions in the trace; second, unlike the Boolean case, we cannot guarantee that if there is no lasso-shape trace, there does not exist an infinite trace satisfying the model (since a real variable may be forced to increase forever). Nevertheless, we find the procedure extremely efficient in the framework of requirements validation.

In order to prove the emptiness of the FTS, we use predicate abstraction. We adopt a CEGAR loop, where the abstraction generation and refinement are completely automated. The loop consists of four phases: 1) *abstraction*, where the abstract system is built according to a given set of predicates; the abstract state space is computing by passing to the SMT solver an ALLSAT problem; 2) *verification*, where the non-emptiness of the language of the abstract system is checked; if the language is empty, it can be concluded that also the concrete system has an empty language; otherwise, an infinite trace is produced; the abstract system is finite so that we can used classical model checking techniques; 3) *simulation*: if the verification produces a trace, the simulation checks whether it is realistic by simulating it on the concrete system; if the trace can be simulated in the concrete system, it is reported as a real witness of the satisfiability of the formula; the trace is simulated by checking the satisfiability of the SMT problem; 4) *refinement*: if the simulation cannot find a concrete trace corresponding to the abstract one, the refinement discovers new predicates that, once added to the abstraction, are sufficient to rule out the unrealistic path; also this step is solved with an SMT solver.

3 The ETCS project

The European Train Control System (ETCS) is a project supported by the European Union aiming at the implementation of a common train control system in all European countries to allow the uninterrupted movement of train across the borders. ETCS is based on the implementation on board (in a suitable computer based equipment) of a set of safety critical functions of speed and distance supervision and of information to the driver. Such functions rely on data transmitted by track-side installations through two communication channels: fixed spot transmission devices, called balises, and continuous, bidirectional data transmission through radio according to the GSM standard. ETCS is already installed in important railway lines in different European countries (like Spain, Italy, The Netherlands, Switzerland) and installations are in progress in other countries, such as Sweden, UK, France, Belgium and also non-European railways such as China, India, Turkey, Arabia, South Korea, Algeria and Mexico.

Since 2005, the European Commission decided to give the European Railway Agency (ERA) the role of system authority for ETCS, with the responsibility of managing the evolution of the specifications (change control management), ensuring their consistency, and guaranteeing the backwards compatibility of new versions with the old ones.

In 2007, ERA issued a call to tender for the development of a methodology complemented by a set of support tools, for the formalization and validation of the ETCS specifications. The activity poses many hard problems. First, the ETCS documents are written in natural language, and may thus contain a high degree of ambiguity. Second, the ETCS specifications are still in progress, and receive contribution by many people with different culture and background. Third, the ETCS comprises a huge set documents, and comes with severe issues of scalability.

The EuRailCheck project, originated from the successful response to the call to tender by the consortium composed by "Registro Italiano Navale (RINA)", a railway certifying body, "Fondazione Bruno Kessler - irst", a research center, and "Dr. Graband and Partners", a railway consultancy company.

Within the project, we developed a support tools, covering the various phases of the described methodology, based on the integration of algorithmic formal verification techniques within traditional design tools.

Moreover, a realistic subset of the specification was formalized and validated applying the developed methodology and tools.

The results of the project were then further exploited and validated by domain experts external to the consortium. The evaluation was carried out in form of a workshop, followed by hands-on training courses. These events were attended by experts from manufacturing and railways companies, who provided positive feedback on the applicability in the large of the methodology.

3.1 Tool support

The EuRailCheck supporting tool, which has been designed and developed within the project, considered several user and technical requirements such as easy of use, and openness.

The technological basis was identified in two tools provided by IBM: the RequisitePro suite was used as a front end for the management of the ETCS informal requirements; and, the Rational Software Architect (RSA) was used for the management of the formalization of the ETCS requirements into UML class diagrams and CNL constraints.

RSA was used for its openness in the manipulation of UML specification, and its customizability thanks to the embedded Eclipse platform it is built upon. RSA was used as a gluing platform, and all the modules were developed as plug-ins for RSA. The main functionalities include RequisitePro custom tagging, annotation of UML diagrams with CNL (syntax checking, completion), support for the instantiation to finite domains, control of the validation procedure. Moreover, we also developed, relying on the API provided by RequisitePro and on the Eclipse platform, the traceability links among the informal requirements classified in RequisitePro and their formal counterpart in UML and CNL inside RSA.

The verification back-end is based on an extended version of the NuSMV/CEGAR [CCGR00] model checker, able to deal with continuous variables, and to analyze temporally complex expressions in RELTL [EF06, CRST09, CRT09].

A view of the software components of EuRailCheck tool is given in Figure 1.

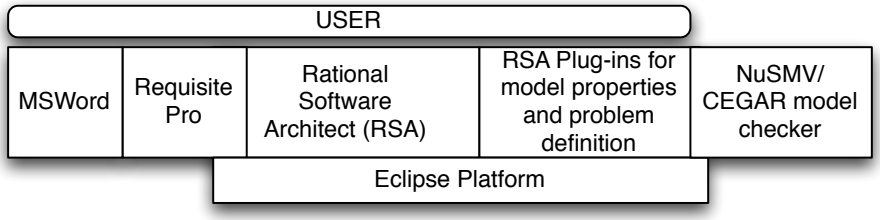


Figure 1: The EuRailCheck architecture.

4 Related work

Several works faced with the problem of the formal specification and validation of requirements. Some of them focused on the problem of formalizing natural language specifications, other focused on the formal specification languages to be used in such a task, other proposed a methodological approach to the requirements representation and validation.

On the first side, works such as [FGR+94] and [AG06] aim at extracting automatically from a natural language description a formal model to be analyzed. However, their target formal languages cannot express temporal constraints over object models. Moreover, they miss a methodology for an adequate formal analysis of the requirements.

Several formal specification languages such as Z [Spi92], B [Abr96], and OCL [OMG06] have been proposed for formal model-based specification. They are very expressive but require a deep background in order to write a correct formalization. Alloy [Jac02] is a formal language for describing structural properties of a system relying on the subset of Z [Spi92] that allows for object modeling. An Alloy specification consists of basic structures representing classes together with constraints and operations describing how the structures change dynamically. Alloy only allows to specify attributes belonging to finite domains (no Reals or Integers). Thus, it would have been impossible to model the Train position as requested by the ETCS specifications. Although Alloy supports the "next" operator ("prime" operator) to specify the temporal evolution of a given object, it does not allow to express properties using LTL and regular expressions.

Among the methodological approaches, in [HJL96], a framework is proposed for the automated checking of requirement specifications expressed in Software Cost Reduction tabular notation, which aims at detecting specification problems such as type errors, missing cases, circular definitions and non-determinism. Although this work has many related points to our approach, the proposed language is not adapt to formalize requirements that contain functional descriptions of the system at high level of abstraction with temporal assumptions on the environment. Formal Tropos (FT) [BGG+04, SPGM05, FLM+04] and KAOS [DDMvL97, vL09] are goal-oriented software development methodologies that provide a visual modeling language that can be used to define an informal specification, allowing to model intentional and social concepts, such as those of actor, goal, and social relationships between actors, and annotate the diagrams with temporal constraints to characterize the valid behaviors of the model. Both in FT and in KAOS a specification consists of a sequence of class declarations such as actors, goals. Each declaration associates a set of attributes to the class. The temporal behavior of the instances is specified by means of temporal constraints expressed in a typed first-order LTL. On the formalization part, our methodology is similar to the FT and the KAOS ones, but it differs in the expressiveness of the formalization language. FT and KAOS are limited to LTL temporal operators, while our approach allows to express constraints and properties with a logic that mixes LTL with regular

expression.

5 Conclusions

In this paper we described a recent research line that we are pursuing in the context of requirement validation for safety-critical applications. We developed an end-to-end methodology for the analysis of requirements, which combines informal and formal techniques. The property-based approach guarantees traceability, by allowing for a direct correspondence between the components of the informal specification and their formalized counterparts. The formal specification language mixes linear-temporal logic with first-order and hybrid components. Automatic albeit incomplete techniques based on model checking are used to check consistency, entailment of required properties, and possibility of desirable scenarios.

The methodology has been applied in a project with industrial partners for the formalization and validation of railways requirements. During the project, we developed a tool that integrates, within a commercial environment for traditional requirements management and model-based design, advanced techniques for formal validation. The tool has been used and validated by potential end users external to the project's consortium.

In the future, we will pursue the following lines of activity. First, we will investigate the application of automated techniques for Natural Language Processing (e.g. automated tag extraction, discourse representation theory), in order to increase the automation of the first phase of the methodology. Second, we will explore extensions to the expressiveness of the formalism, the relative scalability issues of the verification tools. In particular, we want to integrate optimization such as the ones described in [CRST07] and in [Ton09].

References

[Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[AG06] V. Ambriola and V. Gervasi. On the Systematic Analysis of Natural Language Requirements with CIRCE. *Autom. Softw. Eng.*, 13(1):107–167, 2006.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS*, pages 193–207, 1999.

[BGG+04] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[CCGR00] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.

[CCM+09] Roberto Cavada, Alessandro Cimatti, Alessandro Mariotti, Cristian Mattarei, Andrea Micheli, Sergio Mover, Marco Pensallorto, Marco Roveri, Angelo Susi, and Stefano Tonetta. Eurailcheck: Tool support for requirements validation. In *Proceedings of the 24th IEEE/ACM International Conference Automated Software Engineering (ASE 2009)*, 2009. to appear.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, pages 154–169, 2000.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, London, England, 1999. ISBN 0-262-03270-7.

[CRST07] A. Cimatti, M. Roveri, V. Schuppan, and S. Tonetta. Boolean Abstraction for Temporal Logic Satisfiability. In *CAV 2007*, pages 532 – 546, 2007.

[CRST08a] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta. From Informal Requirements to Property-Driven Formal Validation. In *FMICS*, LNCS, L'Aquila, Italy, sep 2008. Springer.

[CRST08b] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta. Object models with temporal constraints. In *SEFM*, pages 249–258. IEEE Computer Society, 2008.

[CRST09] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta. Formalizing requirements with object models and temporal constraints. *Journal of Software and Systems Modeling (SoSyM)*, 2009. DOI 10.1007/s10270-009-0130-7.

[CRT08] A. Cimatti, M. Roveri, and S. Tonetta. PSL Symbolic Compilation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(10):1737–1750, 2008.

[CRT09] A. Cimatti, M. Roveri, and S. Tonetta. Requirements Validation for Hybrid Systems. In *CAV 2009*, LNCS, pages 188–203. Springer, 2009.

[DDMvL97] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: an environment for goal-driven requirements engineering. In *ICSE'97*, pages 612–613. ACM, 1997.

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL.* Springer-Verlag, 2006.

[FGR+94] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting Requirement Formalization by Means of Natural Language Translation. *Formal Methods in System Design*, 4(3):243–263, 1994.

[FLM+04] A. Fuxman, L. Liu, J. Mylopoulos, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. *Requirements Engineering*, 9(2):132–150, 2004.

[HJL96] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

[Jac02] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer, 1992.

[NF96] R. Nelken and N. Francez. Automatic Translation of Natural Language System Specifications. In *CAV*, pages 360–371, 1996.

[OMG06] OMG. *Object Constraint Language: OMG available specification Version 2.0*, 2006.

[SPGM05] A. Susi, A. Perini, P. Giorgini, and J. Mylopoulos. The Tropos Metamodel and its Use. *Informatica*, 29(4):401–408, 2005.

[Spi92] J. M. Spivey. *The Z Notation: a reference manual, 2nd edition*. Prentice Hall, 1992.

[Ton09] S. Tonetta. Abstract Model Checking without Computing the Abstraction. In *FM*, 2009. To appear.

[vL09] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

Flexible Lyapunov Functions and Applications to Fast Mechatronic Systems

Mircea Lazar

Dept. of Electrical Eng., Eindhoven Univ. of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
m.lazar@tue.nl

Abstract. The property that every control system should posses is stability, which translates into safety in real-life applications. A central tool in systems theory for synthesizing control laws that achieve stability are control Lyapunov functions (CLFs). Classically, a CLF enforces that the resulting closed-loop state trajectory is contained within a cone with a fixed, predefined shape, and which is centered at and converges to a desired converging point. However, such a requirement often proves to be overconservative, which is why most of the real-time controllers do not have a stability guarantee. Recently, a novel idea that improves the design of CLFs in terms of flexibility was proposed. The focus of this new approach is on the design of optimization problems that allow certain parameters that define a cone associated with a standard CLF to be decision variables. In this way non-monotonicity of the CLF is explicitly linked with a decision variable that can be optimized on-line. Conservativeness is significantly reduced compared to classical CLFs, which makes *flexible CLFs* more suitable for stabilization of constrained discrete-time nonlinear systems and real-time control. The purpose of this overview is to highlight the potential of flexible CLFs for real-time control of fast mechatronic systems, with sampling periods below one millisecond, which are widely employed in aerospace and automotive applications.

1 Introductory Overview

One of the interesting problems in nonlinear control systems is the synthesis of control laws that achieve stability [1, 2]. Control Lyapunov functions (CLFs) [3, 4] represent a powerful tool for providing a solution to this problem. The classical approach is based on the *off-line* design of an explicit feedback law that renders the derivative of the CLF negative. An alternative to this approach is to construct an optimization problem to be solved *on-line*, such that any of its feasible solutions renders the derivative of a candidate CLF negative. This method can be traced back to the early results presented in [5], followed by the more recent articles [6, 7], where synthesis of CLFs is performed in a receding horizon fashion.

All the above works mainly deal with the continuous-time case, while conditions under which these results can be extended to sampled-data nonlinear

systems using their approximate discrete-time models can be found in [8]. An important article on control Lyapunov functions for discrete-time systems is [9]. Therein, classical continuous-time results regarding existence of CLFs are reproduced for the discrete-time case. A significant relaxation in the *off-line* design of CLFs for discrete-time systems was presented in [10], where parameter dependent quadratic CLFs are introduced. Also, interesting approaches to the off-line construction of Lyapunov functions for stability analysis were recently presented in [11], [12] and [13].

Despite the popularity of CLFs within systems theory there is still a significant gap in the application of CLFs in real-time control in general, and control of fast systems (i.e. systems with a very small sampling interval) in particular. The main reason for this is conservativeness of the sufficient conditions for Lyapunov asymptotic stability which are employed by most off-line and on-line methods for constructing CLFs.

To illustrate this consider the graphical depiction in Figure 1. Classically, a

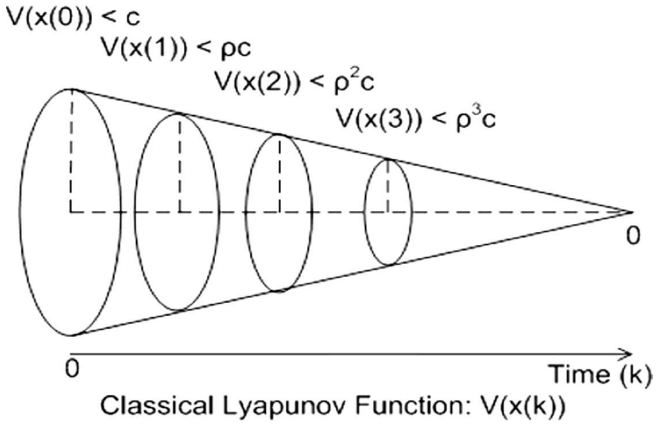


Fig. 1. A graphical illustration of classical CLFs ($\rho \in [0, 1)$, $c \in \mathbb{R}_{>0}$).

CLF enforces that the resulting closed-loop state trajectory is contained within a cone with a fixed, predefined shape, which is centered at and converges to a desired converging point. This cone is obtained by characterizing the evolution of the state via its state-space position at each discrete-time instant, with respect to a corresponding sublevel set of the Lyapunov function. Typical examples of relevant classes of systems for which classical CLFs are overconservative are linear and nonlinear chains of integrators with bounded inputs and state constraints [14] and discontinuous nonlinear and hybrid systems [15]. Furthermore, in many real-life control problems classical CLFs prove to be overconservative. For example, consider the control of a simple electric circuit, such as the Buck-Boost DC-DC converter. At start-up, to drive the output voltage to the reference very fast, the inductor current must rise and stay far away (e.g., 5[A]) from its

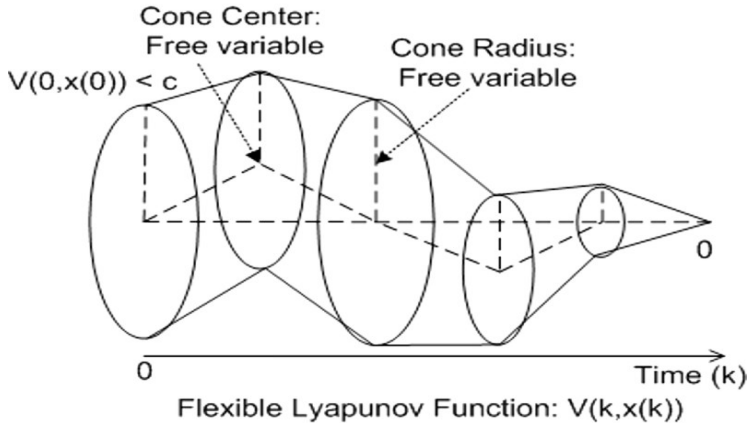


Fig. 2. A graphical illustration of flexible CLFs ($c \in \mathbb{R}_{>0}$).

corresponding steady-state value (e.g., 0.01[A]) for quite some time. Another typical and very relevant real-life example is control of position and speed in mechatronic devices, such as electromagnetic actuators. For a given position reference, the speed must increase very fast at start-up and then return to its steady state value, which is equal to zero. In both cases enforcing a classical CLF design is obviously conservative.

Motivated by such examples, recently, in [16], a methodology that reduces the conservatism of CLF design for discrete-time nonlinear systems was proposed. Rather than searching for a global CLF (i.e. on the whole admissible state-space), therein the focus was on relaxing CLF-type conditions for a predetermined local CLF through on-line optimization problems, as it is graphically illustrated in Figure 2. The goal of this overview is to highlight the potential of flexible CLFs for real-time control of fast mechatronic systems, with sampling periods below one millisecond, which are widely used in aerospace and automotive applications. This includes control of electro-magnetic actuators [17] and a real-time application to the control of DC-DC power converters.

Acknowledgements This research is supported by the Veni grant "Flexible Lyapunov Functions for Real-time Control", grant number 10230, awarded by STW (Dutch Science Foundation) and NWO (The Netherlands Organization for Scientific Research).

References

1. E. D. Sontag, "Stability and stabilization: Discontinuities and the effect of disturbances," in *Nonlinear Analysis, Differential Equations, and Control.* Clarke, F.H., Stern, R.J. (eds.), Kluwer, Dordrecht, 1999, pp. 551–598.
2. P. Kokotović and M. Arcak, "Constructive nonlinear control: a historical perspective," *Automatica*, vol. 37, no. 5, pp. 637–662, 2001.

3. Z. Artstein, "Stabilization with relaxed controls," *Nonlinear Analysis*, vol. 7, pp. 1163–1173, 1983.
4. E. D. Sontag, "A Lyapunov-like characterization of asymptotic controllability," *SIAM Journal of Control and Optimization*, vol. 21, pp. 462–471, 1983.
5. E. Polak and D. Q. Mayne, "Design of nonlinear feedback controllers," *IEEE Transactions on Automatic Control*, vol. AC-26, no. 3, pp. 730–733, 1981.
6. J. A. Primbs, V. Nevistić, and J. C. Doyle, "A receding horizon generalization of pointwise min-norm controllers," *IEEE Transactions on Automatic Control*, vol. 45, no. 5, pp. 898–909, 2000.
7. P. Mhaskar, N. H. El-Farra, and P. D. Christofides, "Stabilization of nonlinear systems with state and control constraints using Lyapunov-based predictive control," *Systems & Control Letters*, vol. 55, pp. 650–659, 2006.
8. L. Grüne and D. Nesić, "Optimization based stabilization of sampled-data nonlinear systems via their approximate discrete-time models," *SIAM Journal of Control and Optimization*, vol. 42, no. 1, pp. 98–122, 2003.
9. C. M. Kellett and A. R. Teel, "Discrete-time asymptotic controllability implies smooth control-Lyapunov function," *Systems & Control Letters*, vol. 52, pp. 349–359, 2004.
10. J. Daafouz, P. Riedinger, and C. Iung, "Stability analysis and control synthesis for switched systems: A switched Lyapunov function approach," *IEEE Transactions on Automatic Control*, vol. 47, pp. 1883–1887, 2002.
11. M. Malisoff and F. Mazenc, "Construction of strict Lyapunov functions for discrete time and hybrid time-varying systems," *Nonnlinear Analysis: Hybrid Systems*, vol. 2, pp. 394–407, 2008.
12. H. Ito, "A degree of flexibility in Lyapunov inequalities for establishing input-to-state stability of interconnected systems," *Automatica*, vol. 44, no. 9, pp. 2340–2346, 2008.
13. A. A. Ahmadi and P. A. Parrilo, "Non-monotonic Lyapunov functions for stability of discrete time nonlinear and switched systems," in *47th IEEE Conference on Decision and Control*, Cancun, Mexico, 2008, pp. 614–621.
14. F. Mazenc, "Stabilization of feedforward systems approximated by a non-linear chain of integrators," *Systems & Control Letters*, vol. 32, pp. 223–229, 1997.
15. M. S. Branicky, V. S. Borkar, and S. K. Mitter, "A unified framework for hybrid control: model and optimal control theory," *IEEE Transactions on Automatic Control*, vol. 43, no. 1, pp. 31–45, 1998.
16. M. Lazar, "Flexible control Lyapunov functions," in *28th American Control Conference*, St. Louis, USA, 2009, pp. 102–107.
17. R. M. Hermans, M. Lazar, S. Di Cairano, and I. V. Kolmanovsky, "Low complexity model predictive control of electromagnetic actuators with a stability guarantee," in *28th American Control Conference*, St. Louis, Missouri, 2009.

An Architecture for Hybrid BDI Continuous Agents for Space Software*

Louise A. Dennis[1], Michael Fisher[1], Nicholas Lincoln[2], Alexei Lisitsa[1], and Sandor M. Veres[2]

[1] Department of Computer Science, University of Liverpool, UK
[2] Department of Engineering, University of Southampton, UK
Contact: L.A.Dennis@liverpool.ac.uk

Abstract. Current approaches to the engineering of space software such as satellite control systems are based around the development of feedback controllers using packages such as MatLab's Simulink toolbox. These provide powerful tools for engineering real time systems that adapt to changes in the environment but are limited when the controller itself needs to be adapted.

We are investigating ways in which ideas from temporal logics and agent programming can be integrated with the use of such control systems to provide a more powerful layer of autonomous decision making. This paper will discuss our initial approaches to the engineering of such systems.

1 Introduction

Modern control systems are limited in their ability to react flexibly and autonomously to changing situations by the complexity inherent in analysing situations where many variables are present.

We are particularly interested in the control of satellite systems. Consider the problem of a single satellite attempting to maintain a geostationary orbit. Current satellite control systems maintain orbits using feedback controllers. These implicitly assume that any errors in the orbit will be minor and easily corrected. In situations where more major errors occur, e.g. caused by thruster malfunction, it is desirable to modify or change the controller. The complexity of the decision task is a challenge to the imperative programming approach.

We approach the problem from the perspective of rational agents and hybrid systems. We consider a satellite to be an *agent* which consists of a discrete (rational decision making) part and a continuous (calculation) part. The discrete part use the *Belief-Desire-Intention* (BDI) theory of agency [4] and governs high level decisions about when to generate new feedback controllers. The continuous, calculational part is used to derive controllers and to calculate information from continuous data which can be used in the decision making process.

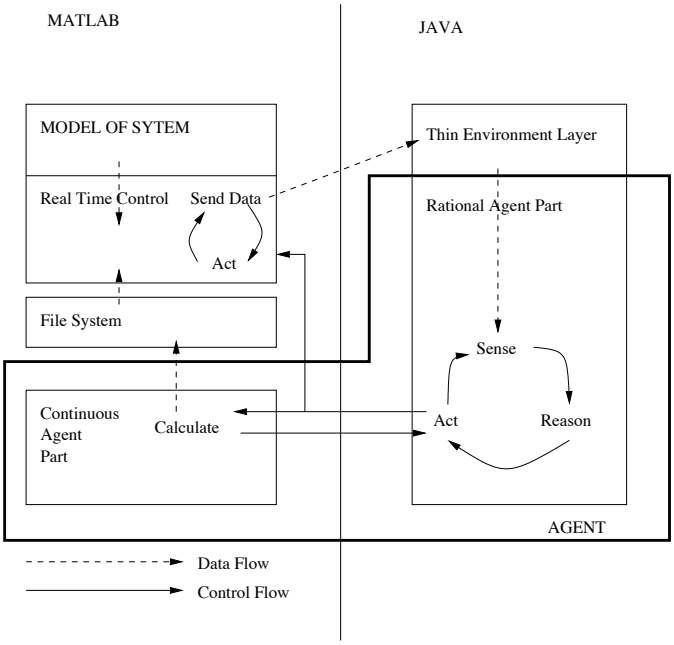


Fig. 1. Implemented Hybrid Agent Architecture

2 Architecture

Our prototype system is shown in figure 1. We have implemented a simulated environment and real time satellite control system in MatLab. The *continuous agent part* is also implemented in MatLab using the Simulink tool kit. MatLab has no easy provision for threaded execution which forces us to use separate instances for the Real Time aspects (i.e. the feedback controller and simulated environment) and for the Continuous Agent part. The agent also contains a *discrete agent part* which is currently implemented in the Gwendolen agent programming language[3]. Gwendolen is implemented on top of JAVA.

The real time control system sends information (which may be pre-processed) to the agent part of the system. When it acts, the discrete part of the agent may either cause the continuous agent part to perform some calculation (and wait for the results) or it may send an instruction to the real time control system to alter its controller. Since the new controller has been created on the fly by the continuous part some aspects of this controller are stored in the shared file system (accessed by both MatLab processes).

⋆ Work funded by EPSRC grants EP/F037201/1 and EP/F037570/1

[3] The choice of language was dictated entirely by convenience. It is a subject for further work to examine more widely used BDI-languages and evaluate which is most appropriate for the system.

3 BDI Programming Aspects

The architecture lets us represent the high-level decision making aspects of the program in terms of the beliefs and goals of the agent and events it observes. So, for instance, when it observes the event that the satellite is in a new position (information relayed to it by the real time controller) the discrete agent part can call on the continuous part to calculate whether this position is within acceptable bounds of the desired orbit (i.e. whether the existing real-time controller is capable of maintaining the orbit). If, as a result of this it gains a belief that the satellite has strayed from the orbit it can request the continuous part to calculate a new path for the satellite to follow using techniques described in [3].

Similarly, if the satellite has strayed from its bounds, the discrete agent part can examine its beliefs about the current status of the thrusters and, if necessary, instruct the continuous part to generate a new feedback controller which takes into account any malfunctions or inefficiencies in the thrusters.

Such programs can be expressed compactly in the BDI-programming style without the need for programming up large decisions trees to consider all possible combinations of thruster status and satellite positions. This should then reduce the possibility for error in the decision-making parts of the program and opens the possibility that existing techniques for model checking such programs [1] can be adapted to verify this part.

4 Future Work

We are currently working on prototype system and case study which will allow us to make comparisons of the agent approach to autonomous decision-making in satellite systems to approaches based on finite state machines. We also are interested in investigating the use of temporal logic and model checking to generate forward planning capabilities for the agent along the lines of those investigated by Kloetzer and Belta [2].

References

1. R. H. Bordini, L. A. Dennis, B. Farwer, and M. Fisher. Automated verification of multi-agent programs. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 69–78, L'Aquila, Italy, September 2008.
2. M. Kloetzer and C. Belta. A Fully Automated Framework for Control of Linear Systems From Temporal Logic Specifications. *IEEE Transactions on Automatic Control*, 53(1):287–297, 2008.
3. N. Lincoln and S. Veres. Components of a Vision Assisted Constrained Autonomous Satellite Formation Flying Control System. *International Journal of Adaptive Control and Signal Processing*, 21(2-3):237–264, October 2006.
4. A. S. Rao and M. Georgeff. BDI Agents: From Theory to Practice. In *Proc. 1st International Conference on Multi-Agent Systems (ICMAS)*, pages 312–319, San Francisco, USA, June 1995.

Synoptic: a Domain Specific Modeling Language for embedded flight-software

A. Cortier, L. Besnard, J.-P. Bodeveix, J. Buisson, F. Dagnat, M. Filali, G. Garcia, T. Gautier, J. Ouy, M. Pantel, A. Rugina, M. Streker, J.-P. Talpin

A. Cortier, J.-P. Bodeveix, M. Filali, M. Pantel, M. Strecker
IRIT-ACADIE, Université Paul Sabatier, 118 Route de Narbonne, F-31062 Toulouse Cedex 9, France
{cortier, bodeveix, filali, pantel, strecker}@irit.fr

L. Besnard, T. Gautier, J. Ouy, J.-P. Talpin
INRIA Rennes - Bretagne Atlantique / IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France
{Loic.Besnard, Thierry.Gautier, Julien.Ouy, Jean-Pierre.Talpin}@irisa.fr,

G. Garcia
Thales Alenia Space, 100 Boulevard Midi, F-06150 Cannes, France
gerald.garcia@thalesaleniaspace.com

A. Rugina
EADS Astrium, 31 rue des Cosmonautes, Z.I. du Palays, F-31402 Toulouse Cedex 4, France
Ana-Elena.RUGINA@astrium.eads.net

J. Buisson, F. Dagnat
Institut Télécom / Télécom Bretagne, Technopôle Brest Iroise, CS83818, F-29238 Brest Cedex 3, France
Fabien.Dagnat@telecom-bretagne.eu

Extended abstract

In collaboration with major European manufacturers, the SPaCIFY project aims at bringing advances in MDE to the satellite flight software industry. It focuses on software development and maintenance phases of satellite lifecycle. The project advocates a top-down approach built on a Domain-Specific Modeling Language (DSML) named Synoptic. The aim of Synoptic is to support all aspects of embedded flight-software design. As such, Synoptic consists of heterogeneous modeling and programming principles defined in collaboration with the industrial partners and end users of the SPaCIFY project.

Used as the central modeling language of the SPaCIFY model driven engineering process, Synoptic allows to describe different layers of abstraction: at the highest level, the software architecture models the functional decomposition of the flight software. This is mapped to a dynamic architecture which defines the thread structure of the software. It consists of a set of threads, where each thread is characterized by properties such as its frequency, its priority and its activation pattern (periodic, sporadic).

Submitted to:
FMA 2009

A mapping establishes a correspondence between the software and the dynamic architecture, by specifying which blocks are executed by which threads. At the lowest level, the hardware architecture permits to define devices (processors, sensors, actuators, busses) and their properties.

Finally, mappings describe the correspondence between the dynamic and hardware architecture on the one hand, by specifying which threads are executed by which processor, and describe a correspondence between the software and hardware architecture on the other hand, by specifying which data is transported by which bus for instance. Figure 1 depicts these layers and mappings.

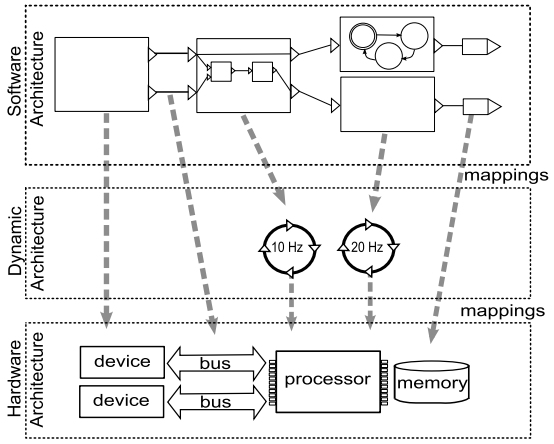


Figure 1: Global view: layers and architecture mappings

The aim is to synthesize as much of this mapping as possible, for example by appealing to internal or external schedulers. However, to allow for human intervention, it is possible to give a fine-grained mapping, thus overriding or bypassing machine-generated schedules. Anyway, consistency of the resulting dynamic architecture is verified by the SPaCIFY tool suite, based on the properties of the software and dynamic model.

At each step of the development process, it is also useful to model different abstraction levels of the system under design inside a same layer (functional, dynamic or hardware architecture). Synoptic offers this capability by providing an incremental design framework and refinement features.

To summarize, Synoptic deals with data-flow diagrams, mode automata, blocks, components, dynamic and hardware architecture, mapping and timing.

The functional part of the Synoptic language allows to model software architecture. The corresponding sub-language is well adapted to model *synchronous islands* and to specify interaction points between these islands and the middleware platform using the concept of *external variables*.

Synchronous islands and middleware form a Globally Asynchronous and Locally Synchronous (GALS) system.

Software architecture The development of the Synoptic software architecture language has been tightly coordinated with the definition of the GeneAuto language [1]. Synoptic uses essentially two types of modules, called blocks in Synoptic, which can be mutually nested: data-flow diagrams and mode automata. Nesting favors a hierarchical design and enables viewing the description at different levels of detail.

By embedding blocks in the states of state machines, one can elegantly model operational modes: each state represents a mode, and transitions correspond to mode changes. In each mode, the system may be composed of other sub-blocks or have different connection patterns among components.

Apart from structural and behavioral aspects, the Synoptic software architecture language allows to define temporal properties of blocks. For instance, a block can be parameterized with a frequency and a worst case execution time which are taken into account in the mapping onto the dynamic architecture.

Synoptic has a formal semantics, defined in terms of the synchronous language SIGNAL [2, 3]. On the one hand, this allows for neat integration of verification environments for ascertaining properties of the system under development. On the other hand, a formal semantics makes it possible to encode the meta-model in a proof assistant. In this sense, Synoptic will profit from the formal correctness proof and subsequent certification of a code generator that is under way in the GeneAuto project.

Synoptic is equipped with an assertion language that allows to state desired properties of the model under development. We are mainly interested in properties that permit to express, for example, coherence of the modes ("if component X is in mode m1, then component Y is in mode m2" or "... can eventually move into mode m2"). Specific transformations extract these properties and pass them to the verification tools.

References

[1] A. Toom, T. Naks, M. Pantel, M. Gandriau and I. Wati: *GeneAuto: An Automatic Code Generator for a safe subset of SimuLink/StateFlow*. European Congress on Embedded Real Time Software (ERTS'08), Société des Ingénieurs de l'Automobile, (2008).

[2] P. Le Guernic, J.-P. Talpin and J.-C. Le Lann: *Polychrony for system design*. Journal for Circuits, Systems and Computers, Special Issue on Application Specific Hardware Design, World Scientific, (2003).

[3] Polychrony and SME. Available at `http://www.irisa.fr/espresso/Polychrony`.

Developing Experimental Models for NASA Missions with ASSL

Emil Vassev and Mike Hinchey
Lero—the Irish Software Engineering Research Centre,
[1]University College Dublin, Ireland
[2]University of Limerick, Ireland
{Emil.Vassev, Mike.Hinchey}@lero.ie

Abstract. NASA's new age of space exploration augurs great promise for deep space exploration missions whereby spacecraft should be independent, autonomous, and smart. Nowadays NASA increasingly relies on the concepts of autonomic computing, exploiting these to increase the survivability of remote missions, particularly when human tending is not feasible. Autonomic computing has been recognized as a promising approach for the development of self-managing spacecraft systems that employ onboard intelligence and rely less on control links. The Autonomic System Specification Language (ASSL) is a framework for formally specifying and generating autonomic systems. As part of long-term research targeted at the development of models for space exploration missions that rely on principles of autonomic computing, we have employed ASSL to develop formal models and generate functional prototypes for NASA missions. This helps to validate features and perform experiments through simulation. Here, we discuss our work on developing such missions with ASSL.

Keywords: space exploration missions, autonomic computing, formal specification, NASA, ASSL

1 Introduction

Autonomic Computing (AC) is an emerging field for the development of large-scale self-managing complex systems [1]. The idea is that complex systems such as spacecraft can autonomously manage themselves and deal with dynamic requirements and unanticipated threats. NASA is currently looking at AC with interest, recognizing in its concepts a promising approach to developing new class of space exploration missions, where spacecraft should be independent, autonomous, and "smart". Autonomic software may make spacecraft capable of planning and executing many activities onboard to meet the requirements of changing objectives and harsh external conditions. Examples of such AC-based unmanned missions are the Autonomous Nano-Technology Swarm (ANTS) concept mission [2] and the Deep Space One mission [1]. The development of systems implying such large-scale automation

implies sophisticated software, which requires new development approaches and new verification techniques.

Practice has shown that traditional development methods cannot guarantee software reliability and prevent software failures, which is very important in complex systems such as spacecraft where software errors can be very expensive and even catastrophic (e.g., the malfunction in the control software of Ariane-5 [3]). Thus, formal methods need to be employed in the development of autonomic spacecraft systems. When employed correctly, formal methods have proven to be an important technique that ensures quality of software [4]. In the course of this research, to develop models for autonomic space exploration missions, we employ the ASSL (Autonomic System Specification Language) formal method. Conceptually, ASSL is an AC-dedicated framework providing a powerful formal notation and computational tools that help developers with problem formation, system design, system analysis and evaluation, and system implementation [5].

2 Preliminaries

2.1 Targeted NASA Missions

Both NASA ANTS and the NASA Voyager missions are targets of this research. Other space-exploration missions, such as NASA Mars Rover and ESA's Herschel are a subject of interest as well.

2.1.1 NASA ANTS

The ANTS (Autonomous Nano-Technology Swarm concept sub-mission PAM (Prospecting Asteroids Mission) is a novel approach to asteroid-belt resource exploration. ANTS provides extremely high autonomy, minimal communication requirements to Earth, and a set of very small explorers with a few consumables [2]. The explorers forming the swarm are pico-class, low-power, and low-weight spacecraft, yet capable of operating as fully autonomous and adaptable agents.

There are three classes of ANTS spacecraft: *rulers*, *messengers* and *workers*. By grouping them in certain ways, ANTS forms teams that explore particular asteroids. The internal organization of a team depends on the task to be performed and on the current environmental conditions. In general, each team has a group leader (ruler), one or more messengers, and a number of workers carrying a specialized instrument. The messengers are needed to connect the team members when they cannot connect directly.

2.1.2 NASA Voyager

The NASA Voyager Mission [6] was designed for exploration of the Solar System. The mission started in 1977, when the twin spacecraft Voyager I and Voyager II were

launched (cf. Figure 1). The original mission objectives were to explore the outer planets of the Solar System. As the Voyagers flew across the Solar System, they took pictures of planets and their satellites and performed close-up studies of Jupiter, Saturn, Uranus, and Neptune.

Fig. 1. Voyager Spacecraft [6].

After successfully accomplishing their initial mission, both Voyagers are now on an extended mission, dubbed the ―Voyager Interstellar Mission. This mission is an attempt to chart the heliopause boundary, where the solar winds and solar magnetic fields meet the so-called *interstellar medium* [7].

2.2 ASSL

The ASSL framework provides a powerful formal notation and suitable mature tool support that allow ASSL specifications to be edited and validated and Java code to be generated from any valid specification [6]. ASSL is based on a specification model exposed over hierarchically organized formalization tiers. This specification model is intended to provide both infrastructure elements and mechanisms needed by an autonomic system (AS). The latter is considered as being composed of special autonomic elements (AEs) interacting over interaction protocols, whose specification is distributed among the ASSL tiers. Note that each tier is intended to describe different aspects of the AS in question, such as *service-level objectives*, *policies*, *interaction protocols*, *events*, *actions*, etc. This helps to specify an AS at different levels of abstraction imposed by the ASSL tiers.

Table 1 presents the multi-tier knowledge model of ASSL. As shown, the latter decomposes an AS in two directions:

1) into levels of functional abstraction;
2) into functionally related tiers (sub-tiers).

Table 1. ASSL Multi-Tier Model

AS	AS Service-Level Objectives	
	AS Self-Management Policies	
	AS Architecture	
	AS Actions	
	AS Events	
	AS Metrics	
ASIP	AS Messages	
	AS Channels	
	AS Functions	
AE	AE Service-Level Objectives	
	AE Self-Management Policies	
	AE Friends	
	AEIP	AE Messages
		AE Channels
		AE Functions
		AE Managed Elements
	AE Recovery Protocols	
	AE Behaviour Models	
	AE Outcomes	
	AE Actions	
	AE Events	
	AE Metrics	

With the first decomposition, an AS is presented from three different perspectives, these depicted as three main tiers (main concepts):

- AS Tier forms a general and global AS perspective exposing the architecture topology, general system behaviour rules, and global *actions*, *events*, and *metrics* applied to these rules.
- ASIP Tier (AS interaction protocol) forms a communication perspective exposing a means of communication for the AS under consideration.
- AE Tier forms a unit-level perspective, where an interacting sets of the AS's individual components is specified. These components are specified as AEs with their own behaviour, which must be synchronized with the behaviour rules from the global AS perspective.

3 Research

In this section, we present our research objectives and current trends.

3.1 Objectives

This research emphasizes the ASSL formal development approach to autonomic systems (ASs). We believe that ASSL may be successfully applied to the development of experimental models for space-exploration missions integrating autonomic features. Thus, we use ASSL to develop experimental models for NASA missions in a stepwise manner (feature by feature) and generate a series of prototypes, which we evaluate in simulated conditions. Here, it is our understanding that both prototyping and formal modeling, which will aid in the design and implementation of real space-exploration missions, are becoming increasingly necessary and important as the urgent need emerges for higher levels of assurance regarding correctness.

3.2 Benefits for Space Systems

Experimental modeling of space-exploration missions can be extremely useful for the design and implementation of such systems. The ability to compare features and issues with actual missions and with hypothesized possible autonomic approaches gives significant benefit. In our approach, we develop space-mission models in a series of incremental and iterative steps where each model includes new autonomic features. This helps to evaluate the performance of each feature and gradually construct a model of more realistic space exploration missions. Different prototypes can be tried and tested (and benchmarked as well), and provide valuable feedback before we implement the real system. Moreover, this approach helps us to discover eventual design flaws in existing missions and the prototype models.

3.3 Modeling NASA ANTS and NASA Voyager with ASSL

ASSL has been successfully used to specify autonomic features and generate prototype models for two NASA projects—the ANTS (Autonomous Nano-Technology Swarm) concept mission (cf. Section 2.1.1) and the Voyager mission (cf. Section 2.1.2). In both cases the generated prototype models helped to simulate space-exploration missions and validate features through simulated experimental results.

In our endeavor to develop NASA missions with ASSL, we emphasized modeling ANTS self-managing policies [8] of self-configuring, self-healing and self-scheduling and the Voyager image-processing autonomic behavior [9]. In addition, we proposed a specification model for the ANTS safety requirements. In general, a complete specification of these autonomic properties requires a two-level approach. They need to be specified at the individual spacecraft level (AE tier) and at the level of the entire system (AS tier). Here, to specify the self-managing policies we used four base ASSL elements:

- *a self-management policy structure* — which describes the self-managing policy under consideration. We use a set of special ASSL constructs such as *fluents* and *mappings* to specify such a policy [5]. With *fluents* we express specific situations, in which the policy is interested, and with *mappings* we map those situations to actions.

- *actions* — a set of actions that can be undertaken by ANTS in response to certain conditions, and according to that policy.
- *events* — a set of events that initiate fluents and are prompted by the actions according to the policies.
- *metrics* — a set of metrics [5] needed by the events and actions.

3.4 Formal Verification

The ASSL framework toolset provides verification mechanisms for *automatic reasoning* about a specified AS. The base validation approach in ASSL comes in the form of *consistency checking*. The latter is a mechanism for verifying ASSL specifications by performing exhaustive traversal to check for both *syntax* and *consistency errors* (type consistency, ambiguous definitions, etc.). In addition this mechanism checks whether a specification conforms to special correctness properties, defined as ASSL semantic definitions. Although considered efficient, the ASSL consistency checking mechanism cannot handle *logical errors* (specification flaws) and thus, it is not able to assert safety (e.g., freedom from deadlock) or liveness properties. Currently, a *model checking* validation mechanism able to handle such errors is under development [10].

4 Conclusion

In this research, we place emphasis on modeling autonomic properties of space-exploration missions with ASSL. With ASSL we model such properties through the specification of self-managing policies and service-level objectives. Formal verification handles consistency flaws and operational Java code is generated for valid models. Generated code is the basis for functional prototypes of space-exploration missions employing self-managing features. Those prototypes may be extremely useful when undertaking further investigation based on practical results and help developers test the autonomic behavior under simulated conditions.

References

1. Murch, R.: Autonomic Computing: On Demand Series. IBM Press, Prentice Hall (2004)
2. Truszkowski, W., Hinchey, M., Rash, J., Rouff, C.: NASA's Swarm Missions: The Challenge of Building Autonomous Software. IT Professional, vol. 6(5), pp. 47–52 (2004)
3. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)
4. Bowen, J.P., Hinchey, M.G.: Ten Commandments of Formal Methods. IEEE Computer, vol. 28(4), pp. 56–63, IEEE Computer Society (1995)

5. Vassev, E., Hinchey, M.: ASSL - A Software Engineering Approach to Autonomic Computing. IEEE Computer, vol. 42(6), pp. 106–109, IEEE Computer Society (2009)
6. The Planetary Society: Space topics: Voyager – The Story of the Mission. http://planetary.org/explore/topics/space_missions/voyager/objectives.html (2009)
7. Jet Propulsion Laboratory: Voyager – The Interstellar Mission. California Institute of Technology, http://voyager.jpl.nasa.gov/ mission/interstellar.html (2009)
8. Vassev, E., Hinchey, M., Paquet, J.: Towards an ASSL Specification Model for NASA Swarm-Based Exploration Missions. Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008) - AC Track, pp.1652-1657, ACM (2008)
9. Vassev, E., Hinchey, M.: Modeling the Image-processing Behavior of the NASA Voyager Mission with ASSL. Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'09), pp. 246-253, IEEE Computer Society (2009)
10. Vassev, E., Hinchey, M., Quigley, A.: Model Checking for Autonomic Systems Specified with ASSL. Proceedings of the First NASA Formal Methods Symposium (NFM 2009), pp.16-25, NASA Technical Report (2009).

Model Checking Nonlinear Hybrid Systems

Pieter Collins, Jan van Schuppen, Bert van Beek,
Davide Bresolin, Ivan Zapreev, Sanja Zivanovic

Centrum Wiskunde & Informatica (CWI), Anmsterdam, The Netherlands

Pieter.Collins@cwi.nl

Abstract

In this talk I will give an overview of model-checking hybrid systems with using the tool Ariadne. I will first give an overview of hybrid systems and the theoretical difficulties involved in their analysis. I will then discuss how hybrid systems are modelled in Ariadne, and the problems that the tool can solve. I will then give an overview of the core functionality of the tool, including the use of Taylor function models. I will give a simple example of safety verification for a nonlinear hybrid system. Finally, I will give perspectives for future development, including support for nondeterministic or stochastic noise.

Abstraction and Verification of Stochastic Hybrid Systems

Alessandro Abate

Delft Center for Systems and Control, TU Delft, The Netherlands

a.abate@tudelft.nl

Abstract

Engineering systems like communication networks or automotive and air traffic control systems, financial and industrial processes like market and manufacturing models, and natural systems like biological and ecological environments exhibit complex behaviors arising from the composition and interactions between their heterogeneous components. Hybrid Systems are mathematical models that are by construction intended to describe such complex systems. The presence of uncertainty, which is virtually unquestionable in many biological systems and often inevitable in engineering systems, naturally leads to the employment of stochastic hybrid models.

This talk will discuss the use of stochastic hybrid models in the verification and control of a large class of engineering and biological systems. In particular, the study will concentrate on understanding the theoretical and computational issues associated with problems of verification of controlled stochastic hybrid systems, and will propose techniques to attain formal abstractions of these models.