EPTCS 184

Proceedings of the

4th International Workshop on

Engineering Safety and Security Systems

Oslo, Norway, June 22, 2015

Edited by: Jun Pang, Yang Liu and Sjouke Mauw

Published: 10th June 2015
DOI: 10.4204/EPTCS.184
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

Jun Pang
University of Luxembourg

Yang Liu
Nanyang Technological University

Sjouke Mauw
University of Luxembourg

The present volume contains the proceedings of the 4th International Workshop on Engineering Safety and Security Systems (ESSS'15). The workshop was held in Oslo, Norway, on June 22nd, 2015, as a satellite event of the 20th International Symposium on Formal Methods (FM'15).

The goal of the workshop was to establish a platform for the exchange of ideas on:

- methods, techniques and tools for system safety and security;
- methods, techniques and tools for analysis, certification, and debugging of complex safety and security systems;
- model-based and verification-based testing;
- emerging application domains such as cloud computing and cyber-physical systems;
- case studies and experience reports on the use of formal methods for analyzing safety and security systems.

This year, the 2nd International Workshop on Safety and Formal Methods (SAFOME'15) was merged into ESSS'15. This resulted in 15 submissions in total, which were all reviewed by at least three referees. After an intensive discussion by the PC members, we selected 6 papers for presentation at the workshop and inclusion in the proceedings. The program of ESSS'15 also included invited talks by Marieke Huisman (University of Twente, Netherlands) and Audun Jøsang (University of Oslo, Norway).

We would like to thank all the authors for submitting their work to ESSS'15 and SAFOME'15 and the members of the Program Committee as well as the external reviewers for their efforts and high-quality reviews. Finally, thanks are due to the organizers of SAFOME'15, Ricardo J. Rodríguez (Universidad de León, Spain) and Stefano Tonetta (Fondazione Bruno Kessler, Italy).

Abstracts of the Invited Talks

- **Speaker**: Marieke Huisman (University of Twente, Netherlands)
  Title: Verification of Concurrent Software
  Abstract: This talk presents the VerCors approach to verification of concurrent software. First we discuss why verification of concurrent software is important, but also challenging, and we show how permission-based separation logic allows one to reason about multithreaded Java programs in a thread-modular way. We discuss in particular how we extend the logic to reason about functional properties in a concurrent setting. Further, we show how the approach is also suited to reason about programs using a different concurrency paradigm, namely kernel programs using the Single Instruction Multiple Data paradigm. Concretely, we illustrate how permission-based separation logic is used to verify functional correctness properties of OpenCL kernels.
- **Speaker**: Audun Jøsang (University of Oslo, Norway)
  Title: Modelling Reliability with Degrees of Uncertainty based on Subjective Logic
  Abstract: System reliability analysis is typically probability-based. However, it is often the case

J. Pang, Y. Liu, and S. Mauw (Eds.): 4th International Workshop
on Engineering Safety and Security Systems 2015 (ESSS'15)
EPTCS 184, 2015, pp. 1–3, doi:10.4204/EPTCS.184.0

that some input arguments are missing or that they are affected by considerable uncertainty, in which case it becomes difficult or impossible to complete the reliability analysis. Subjective logic offers a simple solution whereby probabilistic arguments can be expressed with degrees of uncertainty. This approach can also be used for developing probabilistic formal models of systems, where e.g. assumptions and arguments can be expressed with degrees of uncertainty. This talk presents subjective logic and gives examples of how it can be applied to system reliability analysis.

Program Committee

- Étienne André, University Paris 13, France
- Thomas Arts, Quviq, Sweden
- Guangdong Bai, National University of Singapore, Singapore
- Clara Benac Earle, Universidad Politécnica de Madrid, Spain
- Marius Bozga, VERIMAG, France
- Elena Gómez-Martínez, Universidad Politécnica de Madrid, Spain
- Hans Hansson, Mälardalen University, Sweden
- Marieke Huisman, University of Twente, The Netherlands
- Weiqiang Kong, Dalian University of Technology, China
- Keqin Li, Huawei, Germany
- Yang Liu (PC co-chair), Nanyang Technological University, Singapore
- Sjouke Mauw (PC co-chair), University of Luxembourg, Luxembourg
- Thomas Noll, RWTH Aachen University, Germany
- Peter Csaba Ölveczky, University of Oslo, Norway
- Jun Pang (PC co-chair), University of Luxembourg, Luxembourg
- Cristian Prisacariu, University of Oslo, Norway
- Ricardo J. Rodríguez, Universidad de León, Spain
- Kristin Rozier, NASA/Cincinnati University, USA
- Harald Ruess, Fortiss, Germany
- Wilfried Steiner, TTTech, Austria
- Cong Tian, Xidian University, China
- Stefano Tonetta, Fondazione Bruno Kessler, Italy
- Mohammad Torabi Dashti, ETH Zurich, Switzerland
- Catia Trubiani, Gran Sasso Science Institute, Italy
- Anton Wijs, Eindhoven University of Technology, The Netherlands
- Yoriyuki Yamagata, Japan Advanced Institute of Science and Technology, Japan
- Tian Zhang, Nanjing University, China

External Reviewers

- Alvaro Botas
- Francesco Gallo
- Pingfan Kong
- Sander de Putter
- Ling Shi

Indefinite waitings in MIRELA systems

Johan Arcile Jean-Yves Didier Hanna Klaudel
Laboratoire IBISC, Université d'Evry-Val d'Essonne, France
johan.arcile@ens.univ-evry.fr {jean-yves.didier,hanna.klaudel}@ibisc.fr

Raymond Devillers
Département d'Informatique,
Université Libre de Bruxelles, Belgium
rdevil@ulb.ac.be

Artur Rataj
Institute of Theoretical and Applied Computer Science,
Gliwice, Poland
arturrataj@gmail.com

MIRELA is a high-level language and a rapid prototyping framework dedicated to systems where virtual and digital objects coexist in the same environment and interact in real time. Its semantics is given in the form of networks of timed automata, which can be checked using symbolic methods. This paper shows how to detect various kinds of indefinite waitings in the components of such systems. The method is experimented using the PRISM model checker.

Keywords: mixed reality; timed automata; deadlocks; starvation.

1 Introduction

The aim of this paper is to provide a formal method support for the development of concurrent applications, which consist of components, which mutually interact in a way, that should meet certain real–time constraints, like a reaction time within a given time period. MIRELA (for MIxed REality LAnguage [9, 7, 10, 8]) was initially meant to be used for developing mixed reality (MR) [6] applications, which acquire data from sensors (like cameras, microphones, GPS, haptic arms...), and then distribute it through a shared memory, read by rendering devices, which present the results in a way a human can interpret (using senses like sight, hearing, touch; by highlighting images on a screen, projecting virtual images, mixing virtual and real images, moving robot arms...).

One of the ideas behind MIRELA is to translate a model into an equivalent form understandable by a model checker, as opposed to performing state space exploration like e.g., JPF does [24]. It allows the model to be represented not by a transition matrix, possibly very lengthy and still partial, but by a terse specification in the checker's native input language. The checker may then easily apply symbolic data structures like MTBDDs [16], which may in turn allow for a substantial reduction of the space explosion problem, inherent to an explicit transition matrix.

In order to cope with time constraints when developing MR applications, practitioners rely mostly on fast response and high performance hardware, even if this contradicts other issues, like power saving (hence autonomy) and cost (hence mass production), and does not ascertain that critical constraints will always be respected. Modelling the application before testing it on the actual hardware and validating it by applying formal method techniques to prove its robustness, was the main motivation for the development of the MIRELA framework. It aims at supporting the development process of MR set-ups, which are generally prone to various issues related to time and known to be difficult to control and to adjust. Most mixed reality frameworks, like those cited in [5, 21, 14, 19, 15, 11] do not concentrate on the validation of the developed applications. Some of them emphasise the use of formal descriptions of components in order to enforce a modular decomposition [22, 17], and ease future extensions [18]

J. Pang, Y. Liu, and S. Mauw (Eds.): 4th International Workshop
on Engineering Safety and Security Systems 2015 (ESSS'15)
EPTCS 184, 2015, pp. 5–18, doi:10.4204/EPTCS.184.1

or substitutions of one module by another [13, 12]. Such frameworks do not deal with software failure issues related to time. On the contrary, the main focus of the MIRELA framework is the formal analysis of software failure issues related to time, together with timing performance analyses and the development of automatic tools.

The MIRELA framework [7] proposes a methodology that consists of three phases. In the first phase, a formal specification of the system in the form of a network of timed automata [1] is built. It may be obtained by a translation from a high level description made of connected components [9, 10], and represents an ideal world. The second phase concerns the analysis of the system: it essentially consists in analysing through model-checking a set of desired properties considered important, either the absence of bad behaviours or the satisfaction of timing constraints. In the third phase, such a checked specification is used to produce an implementation skeleton, in the form of a looping controller parametrised with a sampling period and possibly executing several actions in the same period, aiming at preserving those properties [7]. We revisit here essentially the second phase of the methodology of the MIRELA framework. Since the high level specifications of MIRELA are close to a subclass of UPPAAL [23] systems, it was originally considered to use the UPPAAL model-checker to analyse the properties of a MIRELA system [8]. However, a serious problem was faced when trying to detect deadlocks limited to some components, since the UPPAAL query language does not allow nested path quantifiers. A proposed solution was then to use instead the PRISM tool [16] and analyse if and how it may be to detect bad behaviours of a tentative MIRELA system.

The paper is organised as follows. First, we shortly recall the specification language of MIRELA and its semantics in terms of a network of TAST automata (a subclass of timed automata of UPPAAL). Next, various kinds of bad behaviours are defined. Then, Section 3 explains how PRISM may be used to model MIRELA systems, and the next one analyses how to verify the system against the indefinite waitings phenomena. This leads to define a procedure to analyse a MIRELA system, which is illustrated on a well chosen example. Finally we summarise the outcome of this contribution, and comment some future works.

1.1 MIRELA syntax and intuitive semantics

A MIRELA specification [7] (see an example in Figure 1, top left), is defined as a list of component's declarations of the form:

$$\mathit{SpecName}: \quad \mathit{id} = \mathit{Comp}{\rightarrow}\mathit{TList}; \ldots; \ \mathit{id} = \mathit{Comp}{\rightarrow}\mathit{TList}.$$

Each component's declaration $\mathit{Comp}{\rightarrow}\mathit{TList}$ defines a component *Comp* and its target list of components *TList*, which is an optional (comma separated) list of identifiers indicating to which (target) components information is sent, and in which order. Each component also indicates from which (source) components data are expected. A target t of a component c must have c as a source, but it is not required that a source s of a component c has c as an explicit target: missing targets will be implicitly added at the end of the target list, in the order of their occurrence in the specification list. We assume that all the sources of a component are different, and that all the targets of a component are also different[1]. A component *Comp* is either a sensor *Sensor*, a processing unit *PUnit*, a shared memory unit *MUnit* or a rendering

[1]The target list is allowed to be empty; this defines in general a degenerate specification, which may be interesting for technical and practical reasons.

loop *RLoop*, and is specified following the syntax:

$$\begin{array}{lcl} \mathit{Sensor} & ::= & \mathsf{Periodic}(\mathit{min\_start},\mathit{max\_start})[\mathit{min},\mathit{max}] \mid \mathsf{Aperiodic}(\mathit{min\_event}) \\ \mathit{PUnit} & ::= & \mathsf{First}(\mathit{SList}) \mid \mathsf{Both}(\mathit{id},\mathit{id})[\mathit{min},\mathit{max}] \mid \mathsf{Priority}(\mathit{id}[\mathit{min},\mathit{max}],\mathit{id}[\mathit{min},\mathit{max}]) \\ \mathit{MUnit} & ::= & \mathsf{Memory}(\mathit{SList}) \\ \mathit{RLoop} & ::= & \mathsf{Rendering}(\mathit{min\_rg},\mathit{max\_rg})(\mathit{id}[\mathit{min},\mathit{max}]), \end{array}$$

where *SList* is a non empty list of (comma separated) source identifiers of the form $id[min,max]$, indicating that the processing time of data coming from source *id* takes between *min* and *max* time units.

There are two kinds of sensors:

- Periodic ones (e.g., cameras) that need some time for being started (at least *min_start* and at most *max_start* time units), and then capture data periodically, taking between *min* and at most *max* time units for that, and
- Aperiodic ones (e.g., haptic arms or graphical user interfaces) that collect data when an event occurs, the parameter *min_event* indicating the minimal delay between taking two successive events into account.

Processing units process data coming from possibly several different sources of data. They may be combined (in a hierarchy but also in loops) to get more inputs and outputs. Hence the sources are either sensors or processing units, and targets are either memories or processing units. There are the following categories of processing units:

- First: may have one or more inputs (sources) and starts processing when data are received from one of them; the order is irrelevant; if *SList* contains only one element, First is considered as a unary processing unit;
- Both: has exactly two inputs and starts processing when both input data are received, the processing time being between *min* and *max*;
- Priority: has two inputs (master and slave) and starts processing when the master input is ready, possibly using the slave input if it is available before the master one; the duration of processing is in the first time interval $[min,max]$ if the master input is alone available, and in the second time interval $[min,max]$ if both the slave and the master inputs are captured; in figures, the slave input is indicated by a dashed arrow.

A memory access is performed by a rendering loop, a sensor or a processing unit by locking the memory before executing the corresponding task (reading or writing) followed by an unlocking of the memory. A rendering component accesses the memory at a flexible period between *min_rg* and *max_rg* time units, and the processing of data has a duration in the interval $[min,max]$.

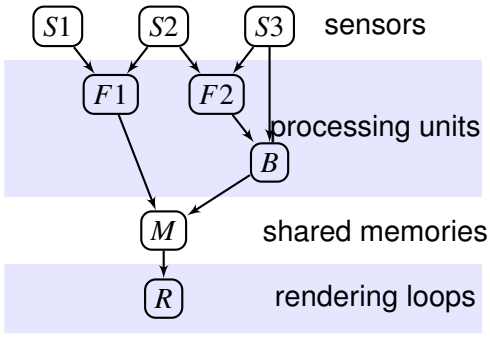
Example 1 *Let us consider the example corresponding to the* MIRELA *system specified in Figure 1 (top left), with three periodic sensors feeding two* First *processing units, and a* Both *one fed by the last sensor and the last* First*, and with a single rendering unit with its associated memory. The corresponding flow of information is illustrated in Figure 1 (top right).* □ 1

Example 2 *This is a variant of Example 1 where* $R = \mathsf{Rendering}(50,75)(M[25,50])$ *is replaced by* $R = \mathsf{Rendering}(75,100)(M[25,50])$*, i.e., the rendering time is longer. The flow of information in this model is the same as in Example 1.* □ 2

Originally, the semantics of a MIRELA specification has been defined and implemented in UPPAAL [23] as a set of timed automata [1, 2, 3, 25] with urgent binary synchronisations, meaning that

Ex_1:

$$
\begin{array}{rcl}
S1 & = & \mathsf{Periodic}(50,75)[75,100];\\
S2 & = & \mathsf{Periodic}(200,300)[350,400]{\rightarrow}(F2,F1);\\
S3 & = & \mathsf{Periodic}(200,300)[350,400]{\rightarrow}(F2,B);\\
F1 & = & \mathsf{First}(S1,S2[50,75]);\\
F2 & = & \mathsf{First}(S2,S3[75,100]);\\
B & = & \mathsf{Both}(S3,F2)[25,50];\\
M & = & \mathsf{Memory}(F1[25,50],B[25,50]);\\
R & = & \mathsf{Rendering}(50,75)(M[25,50]).
\end{array}
$$



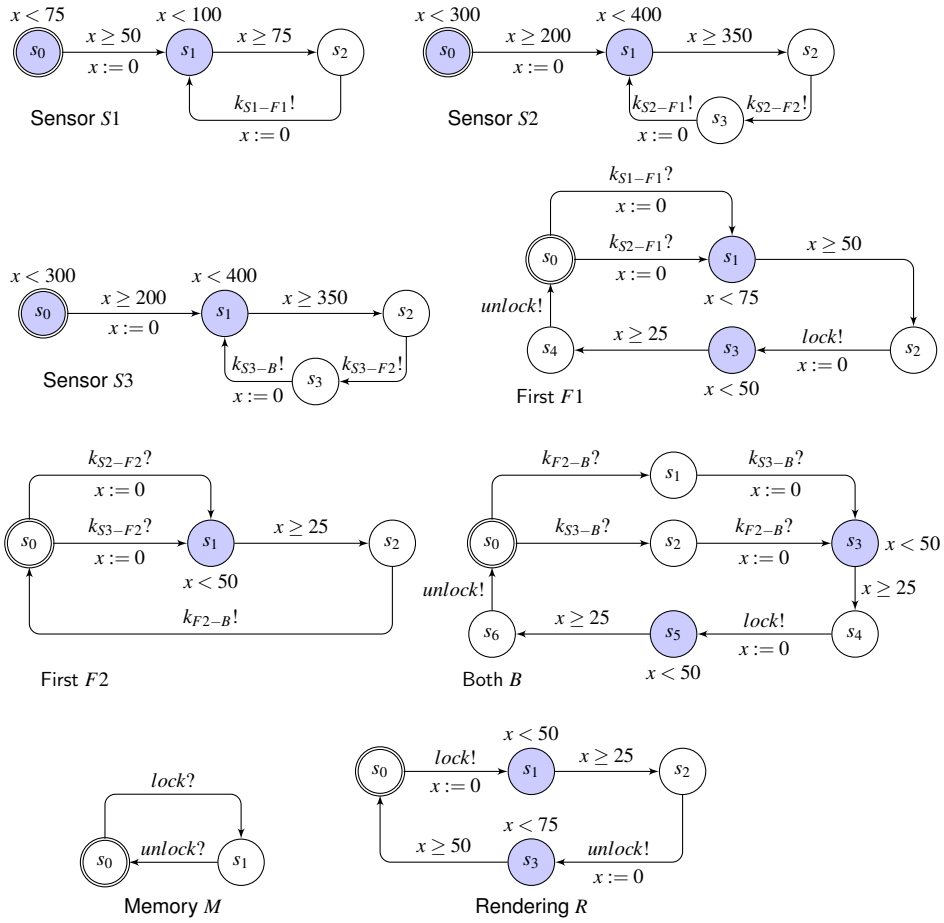


Figure 1: Specification, abstract scheme and TAST representation for Example 1. For Example 2, the TAST representation is as for Example 1 except for the invariant of location s_3 in Rendering R (which becomes $x < 100$) and the guard of its out-going arc (which becomes $x \geq 75$).

when a synchronisation is possible, time may not progress. More precisely, we used a subclass called Timed Automata with Synchronised Tasks (TASTs) in order to cope with implementability issues (see [7] for more details).

Syntactically, a TAST is an annotated directed (and connected) graph, with an initial node, provided with a finite set of non-negative real variables called *clocks* (e.g., x), initially set to 0, increasing with time

and reset ($x := 0$) when needed. Clocks are not allowed to be shared between automata. The nodes (called *locations*) are annotated with *invariants* (predicates allowing to enter or stay in a location, typically either empty (meaning true or $x < \infty$) or of the form $x < e'$, where e' is a natural number. The locations associated with an internal activity (called *activity locations*) are distinguished from the locations where one waits for some event or contextual condition (called *wait locations*). In figures, locations will be represented by round nodes, the initial one having a double boundary, and activity locations are indicated by a coloured background. The arcs are annotated with *guards* (predicates allowing to perform a move) or *communication actions*, and possibly with some clock *resets*. For an activity location, all output arcs have a guard of the form $x \geq e$, all input arcs reset x and the invariant is either empty or of the form $x < e'$, with $0 < e < e'$. For a waiting location, all the output arcs have a communication action of the form $k!$ (output) or $k?$ (input), allowing to glue together the various automata composing a system, since they must occur by input-output pairs. Recall that synchronisations are assumed to be *urgent*, which means that they take place without time progression. In order to structurally avoid Zeno evolutions (i.e., infinite histories taking no time or a finite time), we assume that each loop in the graph of the automaton presents (at least) a constraint $x \geq e$ in a guard (e is strictly positive) and a reset of x for some clock x, or contains only input channels ($k?$).

A TAST representation of Example 1 is depicted in Figure 1 (bottom). The translation from a MIRELA specification to a TAST model (and hence a gateway to the usage of UPPAAL or PRISM for model-checking the system) has been automated by developing a compiler using a parametric approach [4].

2 Bad behaviours

In [8], we analysed the various kinds of bad behaviours that can occur in a timed system in general (and in a MIRELA one in particular). For instance, one may distinguish:

- a *complete blocking* occurs if a state is reached where nothing can happen: no location change is nor will be allowed (because no arc with a true guard is available or the only ones available lead to locations with a non-valid invariant) and the time is blocked (because the invariant of the present location is made false by time passing);
- a *global deadlock* occurs when only time passing is ever allowed: no location change is nor will be possible;
- a *strong (resp. weak) Zeno* situation occurs when infinitely many location changes may be done without time passing (resp. in a finite time delay);
- a *local deadlock* occurs if no location change is available for some component while other components may evolve normally;
- a *starvation* occurs at some point if a component may evolve but the time before may be infinite, because other components may delay it indefinitely;
- an *unbounded waiting* occurs if a component may eventually evolve but the time before is unbounded, because some activity is unbounded.

We shall denote by *indefinite waiting* those last three situations. Note that those situations are not always to be considered as bad: it depends on their semantical interpretation. For instance if a part of a system corresponds to the handling of an error, it may be valid that the system stops after the handling, and it is hoped that it is possible to never reach this situation.

Moreover, we have shown [8] that, for a MIRELA system,

- no (strong or weak) Zeno situation may happen;
- a component may only deadlock in a waiting location;
- a memory unit may only deadlock if all its users deadlock elsewhere;
- a rendering loop may not deadlock, so that a system with a rendering loop may not present a global deadlock.

As a consequence, a global deadlock may not occur in a complete system, i.e., having at least one memory unit and an associated rendering loop; but it can occur in a degenerate (or simplified) system without (memory and) rendering loop. On the contrary, local deadlocks may occur even in complete systems and may propagate to other components. A component may starve for example if it tries to send information to a memory or to a First component which is continually used by other units, and no fairness strategy is applied. From these properties it is sometimes possible to reduce the detection of local deadlocks of a MIRELA system to a global deadlock analysis (easy and efficient with UPPAAL) of a reduced systems, obtained by dropping the memory units and the rendering loops, and the timing constraints as well [8]. However, this does not work in all circumstances and we shall now examine how PRISM may be used for that purpose.

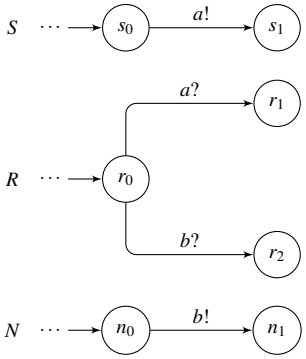
3 PRISM representation of MIRELA

PRISM [16] is a probabilistic model checker intended to analyse a wide variety of systems, including non-deterministic ones. Hence, TASTs and more generally timed automata are particular cases of models PRISM is able to handle. Furthermore, and this is the most interesting feature of PRISM in what we are concerned here, it can use complex (nested) CTL formulas that UPPAAL cannot. However PRISM accepts models that are slightly different from the ones used in UPPAAL. In particular:

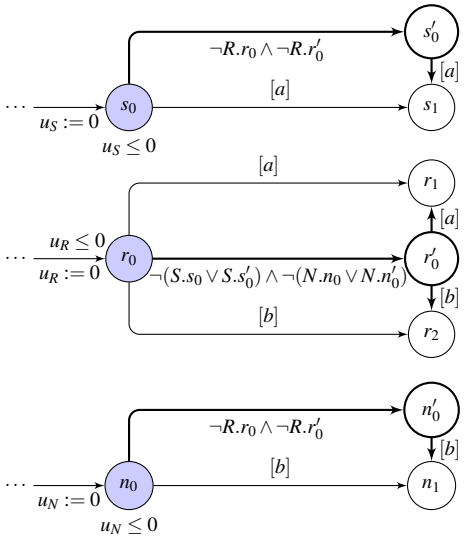
- Communication semantics: in UPPAAL, communications are performed through binary (input/output) synchronisations on some channel k. A synchronisation transition triggers simultaneously exactly one pair of edges $k?$ and $k!$, that are available at the same time in two different components. PRISM implements n-ary synchronisations, where an edge labelled $[k]$ may only occur in simultaneity with edges labelled $[k]$ in all components where they are present;
- Urgent channels: UPPAAL offers a modelling facility by allowing to declare some channels as urgent. Delays must not occur if a synchronisation transition on an urgent channel is enabled. PRISM does not have such a facility and thus it should be "emulated" using a specific construct compliant with PRISM syntax;
- Discrete clocks: the PRISM's tool allowing to check CTL formulas is the *digital clocks engine*, which uses discrete clocks only (and consequently excludes strict inequalities in the logical formulas). This modifies the semantics of the systems, but it may be considered that continuous time, as used by UPPAAL, is a mathematical artefact and that the true evolutions of digital systems are governed by discrete time devices.

Implementing binary communications in PRISM is easy by demultiplying and renaming channels in such a way that a different synchronous channel $[k]$ is attributed to each pair $k?$ and $k!$ of communication labels. In MIRELA specifications, the only labels we have to worry about are the $lock?$ and $unlock?$ labels in each Memory M and the $lock!$ and $unlock!$ labels in the components that communicate with M.

The implementation of urgency is much more intricate, especially if the objective is to be transparent for the execution and also as much as possible for model-checking performance. The solution we adopt



(a) An UPPAAL-style TAST network with urgent channels a and b.



(b) A corresponding PRISM-style network emulating urgency

Figure 2: Urgent communications in PRISM. Thick locations and arcs are the added ones.

here consists in the following construction, illustrated in Figure 2. Let $\mathscr{A} = A_1, \ldots, A_n$ be a network of TAST components A_i. We assume that $\mathscr{A}$ is already renamed in order to implement binary communications. For each $A \in \mathscr{A}$ we declare an additional clock x_A and for each location loc in A with outgoing communication edges to locations $loc_1, \ldots, loc_m$, labelled respectively $[k_1], \ldots, [k_m]$:

- we add the invariant $x_A \leq 0$ to loc and the reset $x_A := 0$ to all input arcs to loc;
- we introduce a new location loc' and an edge from loc to loc' with a guard $\neg(g_{loc}^{k_1} \vee \ldots \vee g_{loc}^{k_m})$, where $g_{loc}^{k_i} \equiv A_j.l_{k_i} \vee A_j.l'_{k_i}$ with $A_j.l_{k_i}$ being the (unique) location with outgoing communication edge labelled $[k_i]$ in some other automaton A_j, and $A_j.l'_{k_i}$ is the corresponding added location;
- for all $i = 1, \ldots, m$, we add an edge from loc' to loc_i, labelled $[k_i]$.

Proposition 1 *The* PRISM *system so constructed presents the same behaviour as the original* TAST *one.*

Proof: (sketch) The general idea behind the construction above is the following: When the control arrives at location loc in some automaton A, since invariant $x_A \leq 0$ on loc requires no time progression, two cases are possible:

- either at least one synchronisation on some $[k_i]$ is immediately possible and one of them must be performed,
- or no synchronisation is possible yet and the control passes to loc' where it may wait as long as one of the synchronisations, let us say $[k_i]$, becomes available.

The latter occurs when the control arrives at a location l having an outgoing arc labelled $[k_i]$, in some automaton A_j. The synchronisation on $[k_i]$ must be performed without time progression due to invariant $x_{A_j} \leq 0$ on l. □ 1

4 Detection of indefinite waitings in MIRELA specifications

In order to detect an indefinite waiting at a location *loc* in a component of the TAST representation $tasts(\mathscr{S})$ of a MIRELA specification $\mathscr{S}$, we may check the CTL formula

$$\phi_{loc} = \text{EF EG } loc,$$

which checks if there exist a path leading to a situation (EF) such that from there it may happen that the component stays (EG) in location *loc*. Since there is no Zeno situation, this may only correspond to an indefinite waiting. If ϕ_{loc} is false, then there is neither a starvation nor an unbounded waiting nor a deadlock in *l*oc. If *l*oc is the activity location of an aperiodic sensor, we know that there is an unbounded waiting, and it is not necessary to perform the model checking for that. The other interesting cases correspond to waiting locations *w*, from which communications *k*! or *k*? only are offered.

If we want to delineate more precisely what happens, we may use a query

$$\psi_w = \text{EF AG } w,$$

which checks if there is a situation where the considered component reached *w* but there is no way to get out of it: this thus corresponds to a local deadlock. From previous observations, it is not useful to apply it to a Memory component if it has corresponding Rendering loop(s), nor to locations waiting for an *unlock*, even if there is no Rendering.

If ϕ_w is true and ψ_w is false, we know that *w* corresponds to a starvation or an unbounded waiting. But if both are true, it may still happen that, while *w* corresponds to a local deadlock for some reachable environment, it is also possible that for another environment, it corresponds to a starvation or an unbounded waiting. This uncertainty may be solved by checking the formula

$$\rho_w = \text{EF EG } (w \wedge (\text{EF } \neg w)),$$

which checks if we can reach a situation where the considered component is in location *w*, it is possible to indefinitely stay in *w* (while other components may progress) but it is also possible to escape from *w*. This corresponds to a starvation situation or an unbounded waiting.

As we mentioned in the introduction, UPPAAL does not support nested CTL queries like ϕ_w, ψ_w and ρ_w. On the contrary, we may check them with PRISM, on the PRISM representation $prism(\mathscr{S})$ of $\mathscr{S}$. Indeed, $prism(\mathscr{S})$ only differs from $tasts(\mathscr{S})$ in that each wait location *w* in $tasts(\mathscr{S})$ is split in two locations *w* and w' in $prism(\mathscr{S})$, such that it is not possible to stay in *w*; if the synchronisation is not performed at *w* with no time progression, $prism(\mathscr{S})$ goes to w', where one shall wait as in *w* in $tasts(\mathscr{S})$. The problem thus comes down to check $\phi_{w'}$, $\psi_{w'}$ and $\rho_{w'}$ on $prism(\mathscr{S})$.

In order to distinguish starvation from unbounded waitings (i.e., if some wait location incurs starvation only, unbounded waiting only, or both in different environments), let us assume the considered specification $\mathscr{S}$ presents *n* aperiodic sensors (with $n > 0$, otherwise there are trivially no unbounded waitings), and let us denote by $a_1, a_2, \ldots, a_n$ their respective initial locations in $tasts(\mathscr{S})$. To check a starvation in location *w*, we may use on $tasts(\mathscr{S})$ the following query formula:

$$\sigma_w = \text{EF EG } (w \wedge (\text{EF } \neg w) \wedge (\text{F } \neg a_1) \wedge \cdots \wedge (\text{F } \neg a_n))$$

which means it is possible to stay indefinitely in *w*, but also to escape from it, without needing that an aperiodic sensor (or many of them) indefinitely stays in its activity location. Hence, if true, this means

there is a pure starvation in w. To check an unbounded waiting in the same location, one may use on $tasts(\mathscr{S})$ the query:

$$\zeta_w = \text{EF}\ ((\text{EG}\ w) \wedge \text{A}((\text{G}\ w) \Rightarrow (\text{FG}\ a_1) \vee \cdots \vee (\text{FG}\ a_n)))$$

which means it is is possible to stay indefinitely in w, but not without being stuck in some a_i at some point. If this is true, this thus means we have an unbounded wait in w. Unfortunately, those last two formulas belong to CTL* and presently, when considering non-deterministic properties, PRISM only supports a fragment of CTL, so that operators G and F must be used in alternation with operators A and E. Hence, σ_w, which contains GF, and ζ_w, which contains FG, are queries that PRISM does not support (yet).

Let us note that, if we were to introduce probabilities in the model, it is very likely that starvations will disappear almost surely (i.e., with probability 1). Indeed, they correspond to the indefinite reproduction of a same kind of finite evolution, and the probability of it is zero, unless that kind of finite evolution has probability 1. Similarly, the probability of unbounded waitings should be zero, like the probability of staying indefinitely in some activity location of an aperiodic sensor (otherwise one could not qualify the waiting as unbounded instead of infinite).

4.1 Procedure and experimental results

From a graph analysis of the specification one may observe that a location in some component may be concerned by starvation, local deadlock or unbounded waiting if it is either a wait location or the initial activity location of an aperiodic sensor. Also, among all the wait locations, we can distinguish the following families:

- the set $\mathscr{N}$ of wait locations that are origins of *unlock*? or *unlock*! transitions: these are concerned by neither local deadlocks nor unbounded waitings nor starvation;
- the set $\mathscr{O}$ of wait locations that are origins of *lock*! transitions: these cannot be concerned by local deadlocks nor by unbounded waitings, but may potentially be concerned by starvation;
- the set $\mathscr{W}$ of all remaining wait locations.

Proposition 2 *Let $\mathscr{S}$ be a* MIRELA *specification.*

1. *If $\mathscr{S}$ comprises an aperiodic sensor, it contains by construction at least a location concerned by an unbounded waiting (which may propagate to other components). On the contrary, if $\mathscr{S}$ has no aperiodic sensor, no unbounded waiting may occur.*
2. *If $\mathscr{S}$ contains an unbounded waiting in some wait location w in $tasts(\mathscr{S})$, $\mathscr{S}$ has aperiodic sensors and $w \in \mathscr{W}$.*
3. *If $\mathscr{S}$ contains a starvation in some wait location w in $tasts(\mathscr{S})$, $w \in \mathscr{O} \cup \mathscr{W}$.*
4. *If $\mathscr{S}$ contains a local deadlock in some wait location w in $tasts(\mathscr{S})$, $w \in \mathscr{W}$.*
5. *A wait location w in $tasts(\mathscr{S})$ incurs a local deadlock, a starvation or an unbounded waiting iff the same occurs in the corresponding location w' in $prism(\mathscr{S})$.*

Proof:

1. By definition, an aperiodic sensor contains a location, in which it may be stuck from the very beginning. It is also the only way to introduce a location where an unbounded waiting is allowed.
2. See the previous point.
3. No location w that is the origin of an *unlock!* or *unlock?* may be concerned by a starvation because this would mean that the memory, once engaged with a rendering or a processing unit, could be indefinitely waiting. As renderings and processing units may never be indefinitely waiting between having performed a *lock!* on the memory and the corresponding *unlock!*, $w \notin \mathscr{N}$. However, one may be indefinitely waiting while trying to perform a *lock!* on a memory or a communication action with a component, but only because the memory (in case of a *lock!*) or the component is continually working for someone else.
4. See the previous points.
5. The only difference in the TAST semantics and the PRISM one is that each wait location w is split into two locations w and w', and it is not possible to stay in w: if the rendez-vous is not performed at w without any delay, PRISM goes to w', where one shall wait as in w in the TAST model.

□ 2

Thus, in order to detect local deadlocks and starvation (or unbounded waitings) in components in MIRELA specifications we propose the procedure described in Algorithm 1, using the PRISM model checker on the PRISM representations $prism(\mathscr{S})$ of MIRELA specifications $\mathscr{S}$.

4.2 Experimental results

We applied this procedure to Examples 1 and 2. In order to automatically translate these examples to the PRISM language, we extended our compiler [20] with the emulation of urgent synchronisations, discussed in Sec. 3, and with a library with definitions of MIRELA components. The results of model checking of formulas and the status of each wait location s_i' are shown in Table 1, where for each component, s_i' is the location added for s_i in Figure 1 in order to emulate urgent communications (see Figure 2). For Example 2, as the model-checking times are similar, we show only locations for which we obtain a different status w.r.t. Example 1.

We may observe that Example 1 presents several locations concerned with both local deadlock and starvation (in different contexts), for which starvation disappears in Example 2 due to the modified timing constraint on the rendering.

5 Conclusions and perspectives

We provided a method allowing to automatically detect indefinite waitings in MIRELA specifications, and to characterise them as local deadlocks, unbounded waitings or starvation problems, or combinations of them. We succeeded thanks to a suitable translation of MIRELA specifications to PRISM, which enabled to model check complex (nested) CTL formulas. An auxiliary but quite general theoretical contribution of the paper is an efficient (and almost transparent) way of expressing urgent communications in PRISM, which was crucial for our first objective.

The translation from MIRELA to TASTs, UPPAAL and PRISM has been automated. The MIRELA compiler can now produce models tuned to specific capabilities of the target model checker. Yet, while we also support PRISM now, we do not take advantage of its major feature of checking models which

Algorithm 1: Determining the status of a wait location

Data: $\mathscr{W}$, $\mathscr{O}$ – sets of wait locations of a MIRELA specification

Result: Compute, for each wait location, if it is a starvation, an unbounded waiting, a (local) deadlock or a combination of them.

```
1  foreach w ∈ 𝒲 ∪ 𝒪 do
2      Check φ_w ← EF EG w;
3      if φ_w = false then
4          w is neither a starvation, unbounded waiting nor deadlock;
5      else
6          if w ∈ 𝒪 then
7              w is a starvation location
8          else
9              Check ψ_w ← EF AG w;
10             if ψ_w = false then
11                 w is a starvation and/or an unbounded waiting
12             else
13                 Check ρ_w ← EF EG (w ∧ (EF ¬w));
14                 if ρ_w = false then
15                     w is a deadlock location
16                 else
17                     w is a local deadlock, a starvation and/or an unbounded waiting location
18                 end
19             end
20         end
21     end
22 end
```

are stochastic. Obviously, probability may allow for much more realistic models and queries within the scope of MIRELA. We plan thus to introduce unreliable and stochastic components, to be checked using formalisms like PCTL, i.e., probabilistic CTL.

The computation times of the example models turned out to be quite reasonable, with the time constants carefully chosen in order to have a large gcd, but we should now consider more complex systems, both in terms of structure, in terms of interval bound characteristics, and in terms of a mixture of stochastic and non-deterministic components. It could also be considered to introduce several traits of actual programming languages, like variables or conditional jumps. In order to stay within the capabilities of model checkers, though, we might e.g., divide a real computer application into functional blocks, each having, beside an actual implementation, a simplified specification for MIRELA, which could be used to e.g., checking properties similar to these discussed here, and consequently discover and analyse possible undesired behaviours of the original application.

Acknowledgment This work has been partly supported by French ANR project SYNBIOTIC and Polish-French project POLONIUM.

example	comp.	w	static set	ϕ_w result	t [s]	ψ_w result	t [s]	ρ_w result	t [s]	status of w
Ex. 1	S1	s'_2	$\mathscr{W}$	false	167					
	S2	s'_2	$\mathscr{W}$	true	228	true	184	**true**	213	D and S
		s'_3	$\mathscr{W}$	true	228	false	156	true	137	S
	S3	s'_2	$\mathscr{W}$	true	226	true	231	**true**	176	D and S
		s'_3	$\mathscr{W}$	false	139					
	F1	s'_0	$\mathscr{W}$	false	123					
		s'_2	$\mathscr{O}$	false	127					
	F2	s'_0	$\mathscr{W}$	false	148					
		s'_2	$\mathscr{W}$	true	214	true	167	**true**	208	D and S
	B	s'_0	$\mathscr{W}$	false	126					
		s'_1	$\mathscr{W}$	true	298	true	198	false	157	D
		s'_2	$\mathscr{W}$	false	137					
		s'_4	$\mathscr{O}$	**true**	202	false	149	**true**	210	S
	R	s'_0	$\mathscr{O}$	false	146					
Ex. 2	S2	s'_2	$\mathscr{W}$	true	181	true	172	**false**	110	D
	S3	s'_2	$\mathscr{W}$	true	208	true	181	**false**	108	D
	F2	s'_2	$\mathscr{W}$	true	163	true	158	**false**	104	D
	B	s'_4	$\mathscr{O}$	**false**	94					

Table 1: Status (D=deadlock, S=starvation) of wait locations in Examples 1 and 2 obtained with Algorithm 1. For Example 2, only properties differing from Example 1 are shown, the mismatching ones are in bold. Model checking times t arisen for a system with AMD Opteron 6234 2.4Ghz and 64GB RAM.

References

[1] Rajeev Alur & David L. Dill (1990): *Automata for modeling real-time systems*. In: *International Colloquium on Algorithms, Languages, and Programming (ICALP) 1990*, *LNCS* 443, Springer, pp. 322–335, doi:10.1007/BFb0032042.

[2] Rajeev Alur & David L. Dill (1991): *The theory of timed automata*. In: *Real Time: Theory in Practice (REX Workshop)*, *LNCS* 600, Springer, pp. 45–73, doi:10.1007/BFb0031987.

[3] Rajeev Alur & David L. Dill (1994): *A theory of timed automata*. *Theoretical Computer Science* 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[4] Johan Arcile (2014): *Implémentation d'un outil de compilation des spécifications MIRELA vers les automates temporisés au format UPPAAL (XML)*. Rapport de stage L3, Département Informatique, Université d'Evry, France.

[5] Martin Bauer, Bernd Bruegge, Gudrun Klinker, Asa MacWilliams, Thomas Reicher, Stephan Riss, Christian Sandor & Martin Wagner (2001): *Design of a Component-Based Augmented Reality Framework*. In: *Proceedings of the International Symposium on Augmented Reality (ISAR)*, doi:10.1109/ISAR.2001.970514.

[6] Mehdi Chouiten, Christophe Domingues, Jean-Yves Didier, Samir Otmane & Malik Mallem (2012): *Distributed mixed reality for remote underwater telerobotics exploration*. In: *Virtual Reality International Conference, VRIC '12*, ACM, France, pp. 1:1–1:6, doi:10.1145/2331714.2331716.

[7] Raymond Devillers, Jean-Yves Didier & Hanna Klaudel (2013): *Implementing Timed Automata Specifications: The" Sandwich" Approach*. In: *13th International Conference on Application of Concurrency to System Design (ACSD), 2013*, IEEE, pp. 226–235, doi:10.1109/ACSD.2013.26.

[8] Raymond Devillers, Jean-Yves Didier, Hanna Klaudel & Johan Arcile (2014): *Deadlock and Temporal Properties Analysis in Mixed Reality Applications*. In: *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, IEEE, pp. 55–65, doi:10.1109/ISSRE.2014.33.

[9] Jean-Yves Didier, Bachir Djafri & Hanna Klaudel (2009): *The MIRELA framework: modeling and analyzing mixed reality applications using timed automata*. *Journal of Virtual Reality and Broadcasting* 6(1).

[10] Jean-Yves Didier, Hanna Klaudel, Mathieu Moine & Raymond Devillers (2013): *An improved approach to build safer mixed reality systems by analysing time constraints*. In: *Proceedings of the 5th Joint Virtual Reality Conference*.

[11] Christoph Endres, Andreas Butz & Asa MacWilliams (2005): *A Survey of Software Infrastructures and Frameworks for Ubiquitous Computing*. *Mobile Information Systems Journal* 1(1), pp. 41–80.

[12] Pablo Figueroa, Walter F Bischof, Pierre Boulanger, H James Hoover & Robyn Taylor (2008): *Intml: A dataflow oriented development system for virtual reality applications*. *Presence: Teleoperators and Virtual Environments* 17(5), pp. 492–511, doi:10.1162/pres.17.5.492.

[13] Pablo Figueroa, J Hoover & Pierre Boulanger (2004): *Intml concepts*. *University of Alberta. Computing Science Department, Tech. Rep.*

[14] Michael Haller, Jürgen Zauner, Werner Hartmann & Thomas Luckeneder (2003): *A generic framework for a training application based on Mixed Reality.* Technical Report, Upper Austria University of Applied Sciences, Hagenberg, Austria.

[15] Charles E Hughes, Christopher B Stapleton, Darin E Hughes & Eileen M Smith (2005): *Mixed reality in education, entertainment, and training*. *Computer Graphics and Applications, IEEE* 25(6), pp. 24–30, doi:10.1109/MCG.2005.139.

[16] M. Kwiatkowska, G. Norman & D. Parker (2004): *Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach*. *International Journal on Software Tools for Technology Transfer (STTT)* 6(2), pp. 128–142, doi:10.1007/s10009-004-0140-2.

[17] Marc Erich Latoschik (2002): *Designing transition networks for multimodal VR-interactions using a markup language*. In: *Proceedings of the 4th IEEE International Conference on Multimodal Interfaces*, IEEE Computer Society, p. 411, doi:10.1109/ICMI.2002.1167030.

[18] David Navarre, Philippe Palanque, Rémi Bastide, Amelie Schyn, Marco Winckler, Luciana P Nedel & Carla MDS Freitas (2005): *A formal description of multimodal interaction techniques for immersive virtual reality applications*. In: *Human-Computer Interaction-INTERACT 2005*, Springer, pp. 170–183, doi:10.1007/11555261_17.

[19] Wayne Piekarski & Bruce H. Thomas (2003): *An Object-Oriented Software Architecture for 3D Mixed Reality Applications*. In: *ISMAR '03: Proceedings of the The 2nd IEEE and ACM International Symposium on Mixed and Augmented Reality*, IEEE Computer Society, Washington, DC, USA, p. 247, doi:10.1109/ISMAR.2003.1240708.

[20] Artur Rataj (2013): *Translation of probabilistic games in J2TADD*. *Theoretical and Applied Informatics* 25(3/4).

[21] Gerhard Reitmayr & Dieter Schmalstieg (2001): *An open software architecture for virtual reality interaction*. In: *Proceedings of the ACM symposium on Virtual reality software and technology*, ACM Press, pp. 47–54, doi:10.1145/505008.505018.

[22] Christian Sandor, Thomas Reicher et al. (2001): *CUIML: A Language for the Generation of Multimodal Human-Computer Interfaces*. In: *Proceedings of the European UIML conference*, 124.

[23] *UPPAAL*. http://www.uppaal.org/.

[24] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park & Flavio Lerda (2003): *Model Checking Programs*. *Automated Software Engineering Journal* 10(2), doi:10.1023/A:1022920129859.

[25] Md Tawhid Bin Waez, Jürgen Dingel & Karen Rudie (2011): *Timed Automata for the Development of Real-Time Systems*. Research Report 2011-579, Queen's University – School of Computing, Canada.

Verification of railway interlocking systems

Simon Busard, Quentin Cappart, Christophe Limbrée,
Charles Pecheur, Pierre Schaus *
Université catholique de Louvain, Louvain-La-Neuve, Belgium
{simon.busard|quentin.cappart|charles.pecheur|pierre.schaus}@uclouvain.be
christophe.limbree@student.uclouvain.be

In the railway domain, an interlocking is a computerised system that controls the railway signalling objects in order to allow a safe operation of the train traffic. Each interlocking makes use of particular data, called application data, that reflects the track layout of the station under control. The verification and validation of the application data are performed manually and is thus error-prone and costly. In this paper, we explain how we built an executable model in NuSMV of a railway interlocking based on the application data. We also detail the tool that we have developed in order to translate the application data into our model automatically. Finally we show how we could verify a realistic set of safety properties on a real-size station model by customizing the existing model-checking algorithm with PyNuSMV a Python library based on NuSMV.

Keywords: Railway interlocking, application data, automatic verification, model checking.

1 Introduction

In the railway domain, an *interlocking* is an arrangement of systems that prevents conflicting train movements in the stations. It is more specially a signalling subsystem that controls the routes, the points and the signals before allowing a train through a station. Computer-based interlockings are configured based on a set of *application data* particular to each station. In this paper, the format considered for the application data is the SSI language [3] that is the electronic interlocking used by the Belgian railways since 1992.

The safety of the train traffic relies on the correctness of the application data. The *European Railway Agency*[1] has edited norms in an effort to harmonize the signalling principles and rules at the European level [14, 5]. Those norms strongly recommend the use of formal methods.

Currently, the application data are prepared manually and are thus subject to human errors. For example, some prerequisite to the clearance (e.g. green light) of the home signal of a route can be missing. This kind of error can easily be discovered by a code review or by testing on a simulator. However, errors caused by concurrent actions (e.g. route commands) are much harder to find. In this case, the combination of possible concurrent actions explodes quickly and testing all possible combinations manually is impracticable.

As testing all the possible scenarios is impossible, the manual validation of the application data relies on a relaxed verification process:

1. The functional tests ensure that the system responds properly to the commands issued by the controller. Those tests are performed by the expert who wrote the application data.

*This research is financed by the Walloon Region as part of the Logistics in Wallonia competitiveness pole.

[1] www.era.europa.eu

J. Pang, Y. Liu, and S. Mauw (Eds.): 4th International Workshop
on Engineering Safety and Security Systems 2015 (ESSS'15)
EPTCS 184, 2015, pp. 19–31, doi:10.4204/EPTCS.184.2

2. The safety tests check that each command (e.g. route) is tested and all the conditions that are supposed to impact the command are tested in all their possible values. Those tests are prepared and carried out by an independent tester.
3. The application data are reviewed by the engineer in charge of the project.

During this process, all anomalies are traced in a bug management tool and must be fixed before the interlocking is commissioned.

1.1 Verification of interlockings using model checking

Our approach to improve the manual verification method is to automatically convert the application data into a model and to verify safety properties on that model with a model checker. A model checker is a tool that automatically checks whether a system meets a given property by comparing the reachable states space of the model of the system and the property. In our case, we used the NuSMV [6] symbolic model checker for which our research team[2] has a broad experience. We also used PyNuSMV, a Python library based on NuSMV that can be used to prototype new model-checking algorithms [4]. PyNuSMV gathers the flexibility of Python and the functionalities of NuSMV in order to efficiently manipulate the BDD data structures.

The safety properties are written based on the track layout of each station and on the safety rules applicable in the signalling domain.

Our approach is divided into the following steps:

1. Generate a model of the interlocking based on the application data. This is done by a translator tool.
2. Generate a model of the trains using a Domain Specific Language that encodes the track layout.
3. State all the properties that must be verified to ensure safety based on the track layout.
4. Combine the models of the interlocking, of the trains, and of the properties into an SMV model that can be processed by NuSMV.
5. Use specific model-checking procedures developed with PyNuSMV to reduce the execution time and produce additional data (route compatibility tables).

This process is shown in Figure 1. Our approach is currently only applicable to a single interlocking. Our other assumptions and abstractions are explained in Section 3.

In the next section, we describe the different components used in our model in order to present our model in Section 3. In Section 4, we explain how our model is constructed based on the application data. In Section 5, we detail the safety properties verified by our model. In Section 6, we discuss how we can improve the performance of the verification. References to related work are provided in Section 7.

2 Interlocking components

Figure 2 shows the track layout of the station of *Namêche*, a Belgian town. This station will be our case study for explaining our approach. The whole station is controlled by a single interlocking that controls 14 routes.

On this figure, the following elements can be identified:

[2] `http://lvl.info.ucl.ac.be`

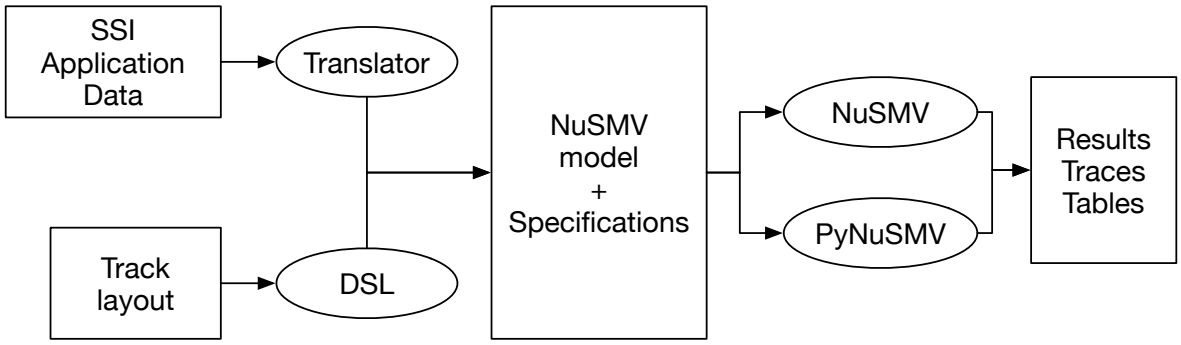


Figure 1: Steps of our approach.

- The track identifiers (e.g. 045).
- The signals (e.g. KM) that are used to grant access to the routes for the trains.
- The points (e.g. P02AM) that are the railway junctions allowing a train to move from one track to another.
- The track circuits[3] (e.g. TC01AM) that are used to detect the vacancy of a portion of the track layout.

The interlocking allows a safe train operation on a railway network or in a station by controlling the *routes*. The routes are the paths followed by the trains when running through a station. For instance, R_KM_045 is a route going from signal KM to track 045. The interlocking handles a route command in the following manner:

1. When a route is requested, it verifies whether the command is safe. This means that the track components (points and track circuits) requested should not be already reserved for another route (the points P01AM, P02BM, P04AM, P04BM, and the tracks TC01AM, TC02BM, TC04BM for R_KM_045).
2. It commands the points by controlling their actuators (points P01AM, P02BM, P04AM, P04BM to the right positions for R_KM_045).
3. It verifies the new status of the points by comparing the command and the replied status of the actuators.
4. It then grants access to the train on the route, setting the origin signal of the route to green (KM for R_KM_045).

A route is composed of several segments called subroutes, corresponding to its track segments (three for route R_KM_045). Each of them is locked when the route is set and is released when the train has fully freed the home track circuit of the subroute, releasing the corresponding points.

This process also makes use of other logical components not shown in Figure 2 like the component materialising a point locking (*UIR*) or the component recording the train passage on the route (*TISP*). The list of controls and verifications stated above are loaded from the application data and used by the interlocking for every route. The fact that the application data properly reflects the track layout and the signalling principles is thus crucial in the safety that the interlocking can achieve. That is why so much effort is devoted to their verification.

[3]Track circuits are sometimes called track segments.

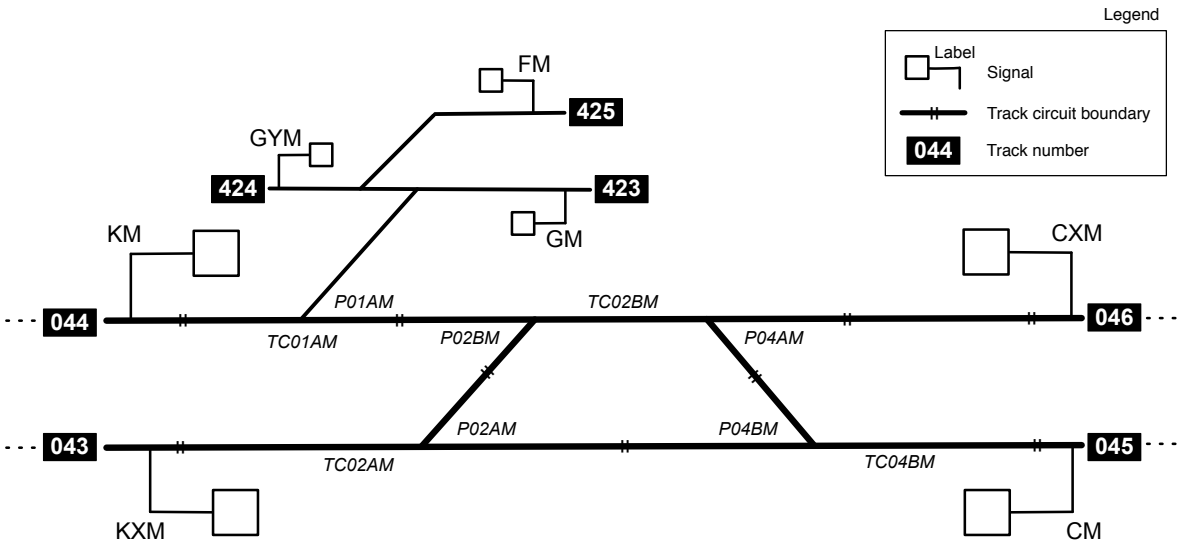


Figure 2: Layout of the Namêche station.

3 Model description

In this section, we describe how the model is designed in order to verify the application data. The complete model can be downloaded from url: `http://lvl.info.ucl.ac.be/Tools/InterlockingModel`.

In order to reduce the size of the state space, several assumptions and abstractions were made:

1. Our method is only applicable to areas controlled by a single interlocking. The case of interlockings interconnected in a network will be studied in our future works.
2. Only two signal aspects are modelled: proceed (green), and danger (red). The trains are supposed to obey the indication given by the signal.
3. The level crossing control and its interaction with the routes is not modelled.
4. The different types of directional locking are not modelled. The directional locking is the mutual exclusion mechanism put in place to prevent head-to-head collisions.
5. The trains can postpone their start when in front of a signal at proceeding aspect but never stop afterwards. The train speed is not modelled.

Our interlocking (SSI) is route based which means:

- A route must be successfully controlled by the controller before a train can run through the station.
- The routes interact with the track side components (e.g.: points, signals).

- The routes using shared resources (e.g.: points) make use of locking variables in order to prevent collisions.
- The path followed by the train is based on the status of the track side components controlled by the routes.

The Figure 3 shows how central the idea of route is in our model. The model is decomposed into NuSMV modules: the interlocking modules and the simulation modules. White modules model the interlocking software components while gray modules model components added to interact with the interlocking.

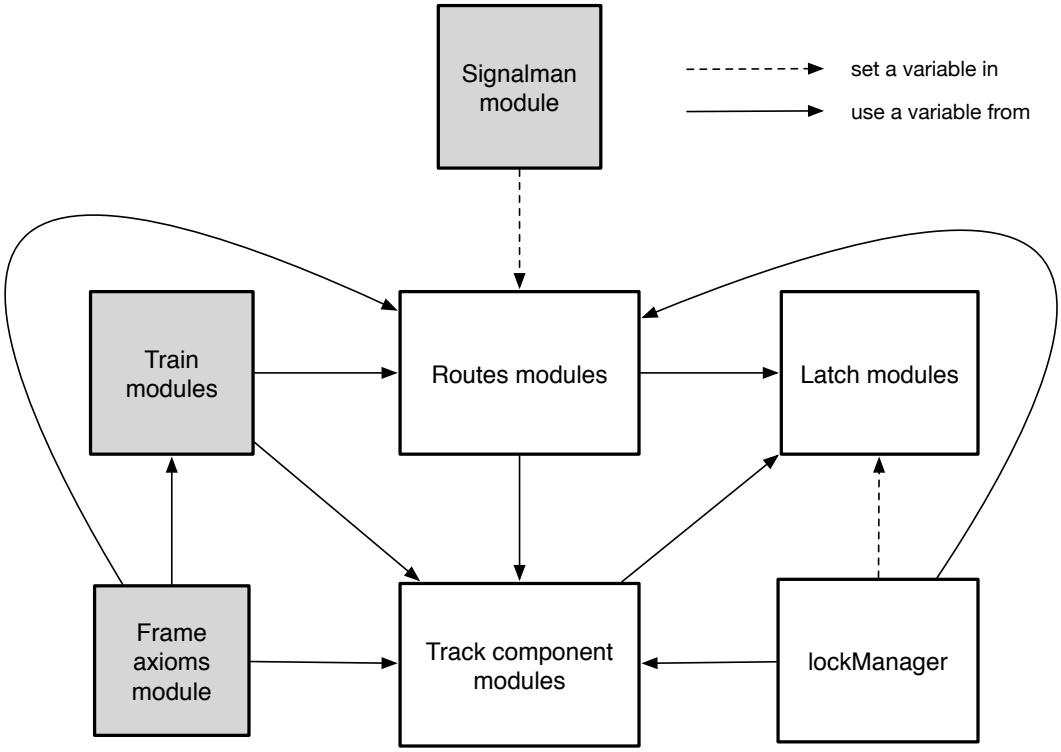


Figure 3: Modules view of the model.

Latch modules: The latches are the global variables shared by the route modules, the point modules, and the lock manager module. The UIR variable is an example of a latch. It is used to lock a point when it is part of a commanded route. The module acts as a record which state is updated by the lock manager module.

Track component modules: The track modules represent a physical component controlled by the interlocking:

- The track segments that hold the state of a track (occupied or clear).
- The points are commanded to the left, or the right position according to the route.

Lock manager module: This module assume the task of locking and unlocking the subroutes and the points when a route is commanded and ran through by a train. This module is a straight emanation from the application data.

Controller module: This module simulates the behaviour of a human commanding the routes. It also ensures that only one route command is issued per transition of the whole system. This module ensures that this behaviour is not violated.

Train module: This module is used to simulate the movement of a train over the adjacent track segments forming the track layout of the station. The track layout is first encoded by mean of a DSL listing all the components like the signals, the track circuits, and the switches. Each component is linked to its siblings taking into account the train direction, and the position of the switches ahead of the train. The graph of the station is then built automatically and all the possible successive train positions are translated into the transitions of the NuSMV module allowing the train simulation.

Route modules: The route lifecycle is described in Section 2. The route modules are a straight translation of the application data from the SSI language to NuSMV. The state machine of a route includes the following states: idle, commanded, proved, and occupied by a train.

Frame axioms module: This module performs three different tasks:

- Changing the status of the track components according to the train movements.
- Triggering a wheel detector when a track segment is occupied.
- Updating the point position after a command.

This module depends on both the application data (routes) and the track layout (trains) to know when the actions must be done and what are the modifications to do.

Given that we want to verify the consistency between the application data and the real track layout, we have to consider a separate source for the application data and the layout. Therefore, unlike the other modules, the train module is not generated from the application data.

Put together, these modules constitute a model simulating the behaviour of an interlocking system as described in the application data and the behaviour of trains according to the track layout. On this model, we can assert and automatically check safety properties with respect to the application data. These properties can be expressed on the state of the trains. For example, a collision occurs if two trains are both located on the same segment. For instance, in Figure 2, such a collision will occur if the application data could allow routes R_KM_045 and R_CM_044 to be set together.

4 Automatic translation of application data

Among all the application data, only a subset is necessary to verify the security of an interlocking system. The rest is either not related to the security or abstracted in our model. Let us now describe the application data used in our model.

Each point can move under a set of conditions. Listing 1 shows how these conditions are represented in the SSI code for a particular point. There are two positions for a point: normal and reverse.[4] Here, the

[4]Normal stands for left and reverse for right.

point P_01AM can be set in a normal position (P_01AM**N**) only if it is is free to move (U_IR(01AM) f). There is a similar rule for the reverse position (P_01AM**R**) .

```
1 *P_01AMN   U_IR(01AM) f /* condition for normal position */
2 *P_01AMR   U_IR(01AM) f /* condition for reverse position */
```

Listing 1: SSI code: Conditions allowing a point to move.

Each route has a set of conditions under which the route request can be granted, and a set of actions that have to be done to fulfil the request. For example, Listing 2 states that the route from Signal CM to Track 044 can only be set if it is not already set (line 2) and if the points are free to be commanded and moved to a certain position (lines 3 and 4). The resulting actions are the setting of the route (line 6), the command of the points (lines 7 and 8) and the locking of the points (line 9). The route and the components requested can be seen on Figure 2.

```
1 *Q_R(CM_044) /* Request for the route CM_044 */
2    if     R_CM_044 xs,
3           P_01AM cfr, P_02BM cfr, P_04AM cfr,
4           P_04BM cfr, P_01BM cfr, P_02AM cfr,
5           U_IR(01AM) f, U_IR(02BM) f, U_IR(04BM) f
6    then   R_CM_044 s
7           P_01AM cr, P_02BM cr, P_04AM cr,
8           P_04BM cr, P_01BM cr, P_02AM cr,
9           U_IR(01AM) l, U_IR(02BM) l, U_IR(04BM) l
```

Listing 2: SSI code: Request for setting a route.

After being locked for a route, the different track components must be freed. According to Listing 3, the subroute U_04M_CM can only be freed if the subroute U_07M_04M is free and if the track T_04BM is clear. There is a set of similar rules for the liberation of the other subroutes and for other components.

```
1 U_04M_CM f if U_07M_04M f, T_04BM c
```

Listing 3: SSI code: Freeing a subroute.

All these data are used to build the NuSMV model. Each interlocking system has its own application data. In other words, we need to build a particular model for each interlocking system. To overcome this issue, we designed a translator which automatically parses the application data and generates the NuSMV model. In this way, we can directly obtain an executable model for each interlocking system. For instance, the NuSMV module corresponding to the route request of Listing 2 is showed on Listing 4.

```
1  MODULE R_CM_044(mainP)
2  VAR
3  cmd : boolean;
4  state : {s, xs}; -- set or unset
5  ASSIGN
6  init(cmd) := FALSE;
7  init(state) := xs;
8  next(state) :=
9     case
10       -- conditions to set the route
11       state = xs & cmd &  -- route not already set
12       mainP.UIR_04BM.st = f & -- track component is free
```

```
13        mainP.UIR_02BM.st = f &
14        mainP.UIR_01AM.st = f &
15        mainP.P_01AM.cfr & -- free to go to reverse position
16        mainP.P_02BM.cfr &
17        mainP.P_04AM.cfr &
18        mainP.P_04BM.cfr &
19        mainP.P_01BM.cfr &
20        mainP.P_02AM.cfr : s;
21        -- conditions to release the route
22        (...)
23    esac;
24 (...)
```

Listing 4: A route command in the NuSMV model

As we can see, this module contains the necessary conditions to set the route. Let us note that the examples presented here do not show all the application data used in our model; other structures such as the train detectors are also used.

5 Safety properties

The safety properties verified on our model are expressed by mean of invariants (properties that are true in any state of the system) and CTL (Computation Tree Logic) formulas. A first set of properties is used to verify that the interaction between the interlocking and the train never ends up in an unsafe sequence causing train collisions or derailments. A second set of properties is used to detect which are the errors in the application data leading to an unsafe behaviour of the interlocking.

Listing 5 shows a sample of properties covering these sets, for route R_CM_044 of the Namêche model.

```
1  INVARSPEC ! (train_1.front = derailed | train_2.front = derailed)
2  INVARSPEC ! (train_1.front = train_2.front)
3  INVARSPEC ! ((train_1.T_01AM | train_2.T_01AM) & P_01AM.willMove)
4  INVARSPEC ! (R_CM_044.st =  s & R_KM_045.st = s)
5  INVARSPEC ! (U_CM_04M.st = l & U_04M_CM.st = l)
6  CTLSPEC AG (T_04BM.st = o & TRP_CM.krc = s -> AX (!R_CM_043.L_CS & !R_CM_044.
       L_CS))
7  INVARSPEC ! (R_CM_044.st = s & U_CM_04M.st = l  & U_04M_07M.st = f)
8  INVARSPEC ! (UIR_01AM.st = l & P_01AM.willMove)
9  INVARSPEC (P_01AM.cmd = P_01BM.cmd)
10 INVARSPEC (R_CM_044.L_CS -> (T_04BM.st = c & T_02BM.st = c & T_01AM.st = c))
```

Listing 5: Safety properties for the Namêche model

Safe interaction between the interlocking and the train This first set embeds the properties verifying that an active simulation of two trains running through the network controlled by the interlocking does not lead to unsafe situations.

- Line 1: Trains never derail. Trains derail when entering a point not set in a position allowing the train to continue its path.
- Line 2: Trains never collide. The property is expressed by stating that the heads of the trains cannot occupy the same position at the same time.
- Line 3: A point (P_01AM) is not allowed to move when its home track-circuit is occupied.

Application data correctness A mistake or an omission in the application data causes the violation of some properties in the first set (e.g. a train collision). However, given a trace leading to a train collision, the identification of which part of the application data is faulty is not trivial. Each property of this second set concerns a route, a part of a route, or a point. As a result, finding the faulty part of the application data in case of violation is easier, as explained hereunder.

- Line 4: Incompatible routes (R_KM_045 and R_CM_044) are never enabled at the same time.
- Line 5: Subroutes in opposite directions (U_CM_04M and U_04M_CM) are never locked at the same time.
- Line 6: The origin signal of a route (R_CM_043 or R_CM_044) immediately goes back to danger after the train has started to run through it. The formula states that in all states (AG) where the train is occupying the track-circuit and has activated the passage detector, the signal is closed in the next state for all possible executions (AX).
- Line 7: The subroutes are released in correct order. In this case, subroute U_04M_07M is not released before subroute U_CM_04M.
- Line 8: A point (P_01AM) will not move when its locking variable (UIR_01AM) is set.
- Line 9: Connecting points (P0_1AM and P_01BM) are always commanded to the same position.
- Line 10: In order to clear the origin signal of a route (R_CM_044.L_CS), all the track circuits must be clear (not occupied by a train).

6 Verification of properties

The model of Namêche station comprises 14 routes, 7 points, and 7 track-circuits for which 132 invariants and 7 CTL formulas were written. These formulas were written manually based on the track layout of Figure 2 for the sake of independence from the application data.

In order to test our model and the adequacy of the properties used on it, we have introduced errors in the application data. The violations were successfully detected and traces were generated. For instance, assigning a wrong locking variable to Point P_01AM can lead to a situation where it is moved under a running train, failing Property 3 in Listing 5. As another example, a missing locking of subroute U_CM_04M leads to a collision between two trains, one going from Signal CM to Point P_04BM and one going from P_04BM to CM, detected by Property 2 in Listing 5.

If we only consider the invariants, the verification with NuSMV takes less than fifteen minutes. However, with CTL formulas it takes few days to complete. To overcome this problem, we implemented custom model-checking algorithms with PyNuSMV.

First, our model includes fairness constraints[5] used to force the trains to eventually progress on the tracks. The trains are modelled such that they can choose to wait at a green signal for an arbitrary long time; fairness constraints are specified to ensure that we consider only executions of the model along which the trains eventually choose to cross a green signal. NuSMV performs model checking of CTL with fairness, which incurs additional computations of fair states and in the verification of some (liveness) formulas. However, the CTL formulas verified on the model are not impacted by the presence of these fairness constraints because all reachable states of the model are fair and no liveness formula is verified.

[5] In the framework of CTL, fairness constraints are sets of states that must be met infinitely often along executions of interest [7].

Thus, to accelerate the verification, standard BDD-based algorithms for CTL without fairness constraints have been used within PyNuSMV instead of those of NuSMV (see for example [7] for more information on these standard algorithms).

Furthermore, these CTL formulas follow the pattern $AG\phi$, where ϕ is a CTL sub-formula, expressing that ϕ is true in all reachable states. NuSMV verifies such formulas by starting from the BDD representing the set of states satisfying ϕ and performing a backward traversal of the system to accumulate the states satisfying the property; then NuSMV compares these states with the BDD of initial states. Nevertheless, this particular case can be improved by checking that all the reachable states satisfy ϕ. This comparison is performed by comparing the BDD of the states satisfying ϕ and the BDD of reachable states, requiring one operation on the two BDDs, instead of the fix-point computation performed by the standard algorithm implemented in NuSMV. Note that this custom approach needs to compute the BDD representing the set of reachable states, but this BDD must be computed to verify the invariants. This BDD is computed by NuSMV itself by performing a standard forward traversal of the model.

Thanks to these custom algorithms, it has been possible to verify the 7 CTL formulas of the previous section within 10 hours, while NuSMV took about 100 hours to verify them. Figure 4 recaps the execution time in function of the number of routes considered.

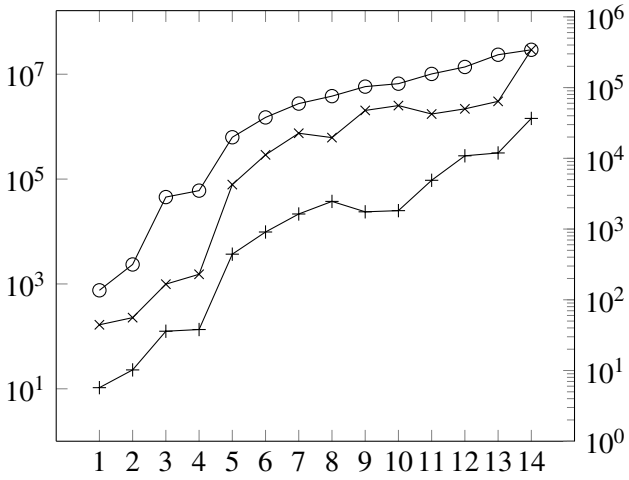


Figure 4: Evolution of number of reachable states ($\circ$) on the left y-axis and verification time for NuSMV ($\times$) and PyNuSMV ($+$) on the right y-axis (in seconds) in terms of number of considered routes.

In order to assess the performance of the custom model-checking algorithms, we have also compared the verification time needed by both tools for models with a reduced number of routes. As we can see in Figure 4, PyNuSMV reduces drastically the execution time when many routes are involved. More precisely, Figure 4 shows the time needed to verify the properties of the previous section with NuSMV and PyNuSMV when the number of considered routes increases. It shows that NuSMV and PyNuSMV behave in similar ways, but PyNuSMV gains an order of magnitude by using custom algorithms.

PyNuSMV can also be used to extract from the model the set of compatible routes. We say that a route is compatible with another route if they can be commanded at the same time, that is, if a train can pass through the first one while another train passes through the second one. Table 1 shows the compatibility table for Namêche (Figure 2). It shows, for example, that Route R_CM_043 and Route R_CXM_044 are compatible while Routes R_KM_045 and R_KM_046 are not. This means that the interlocking system works such that whenever Route R_KM_045 is set, Route R_KM_046 cannot be commanded.

	R_CM_044	R_CM_043	R_CXM_044	R_CXM_043	R_FM_424	R_GM_424	R_GM_044	R_GYM_423	R_GYM_425	R_KM_045	R_KM_046	R_KM_423	R_KXM_045	R_KXM_046
R_CM_044					V	V		V	V					
R_CM_043			V		V	V	V	V	V		V	V		
R_CXM_044					V	V		V	V				V	
R_CXM_043					V	V	V	V	V			V		
R_FM_424							V			V	V	V	V	V
R_GM_424										V	V		V	V
R_GM_044									V				V	V
R_GYM_423										V	V		V	V
R_GYM_425										V	V	V	V	V
R_KM_045														
R_KM_046													V	
R_KM_423													V	V
R_KXM_045														
R_KXM_046														

Table 1: The compatibility table of the station of *Namêche*.

The value *V* in the table means that the corresponding routes are compatible, otherwise they are not compatible. Given that the table is symmetric, only the top half is presented. Thanks to PyNuSMV, such a compatibility table can be extracted by inspection of the set of reachable states. This table can then be used to check that the routes that should not be compatible are not, giving essential information on the application data under interest. Compatibility sets of more than two routes can be produced in the same way: for the Namêche station, 32 sets of three compatible routes exist (e.g. Routes R_CM_043, R_GYM_425 and R_KM_423 can be commanded at the same time), but no set of more than three compatible routes exists.

7 Related work

In [9], Huber and King demonstrates how five vital safety properties can be verified automatically on SSI application data. They implemented a model checker for Geographic Data by replacing the parser and compiler of NuSMV. The resulting tool, gdlSMV, directly reads Geographic Data and builds a corresponding representation on which model checking is performed using NuSMV's symbolic model checking algorithms. In [10], Mirabadi and Yazdi also use the NuSMV model checker and implement a control table verifier that analyses the contents of control table besides the safe train movement conditions and checks for any conflicting settings in the table. In [15], Winter and Robinson modelled the interlocking by means of the formal notation ASM that are more readable. The formal model is translated in NuSMV code and the Safety requirements are expressed in CTL.

In [13, 11, 12], Moller, Nga Nguyen, Roggenbach, Schneider and Treharne propose to combine the state-based and the event-based (a train passing) approaches by using CSP||B. The overall specification combines two communicating models, one made of CSP process descriptions and one made of a collection of B machines. They also propose the OnTrack tool-set that automates workflows for rail-

way verification, starting with graphical scheme plans and finishing with automatically generated formal models set up for verification. In [2], Abo and Voisin explain how Systerel uses the B language and the OVADO tool to verify large interlocking application data set.

In [8], Fantechi, Fokkink and Morzenti give an overview of the trend in railway interlocking verification.

Compared with the previous works, we presented here an unified approach aiming to verify completely the safety of an interlocking system using model checking. More concretely, our approach has the following features:

- A verification of the correctness of the application data.
- A verification of their consistency with the track layout.
- An automatic generation of the models used for the the verification from the application data.
- A Domain Specific Language used to easily encode a track layout into the models.

Taken separately, such features have already been discussed and considered. But to the best of our knowledge, there is no work that merges all of them into a single framework.

8 Conclusions and future work

In this paper, we have explained how we built a model of a railway interlocking in order to verify the correctness of its application data. We have explained how each module of the model can be automatically generated based on the application data by our generator. We have given the list of safety properties that were verified on our model. Those safety properties are designed to cover the tests and verifications that are currently performed manually on the application data. We have shown that the verification of those properties by a model checker brings improvement in the safety and in the efficiency of the verification process ruling the validation of the application data. Finally, we have shown that the verification of large amount of properties (137) is feasible on a realistic size interlocking by mean of custom algorithms based on PyNuSMV.

In our future work, we will focus on the automatic generation of the safety properties based on the track layout. The aim is to use a railway description language such as RailML [1] from which we can generate the safety rules. Furthermore, the properties we verify on the model have been limited to invariants and safety properties; verifying liveness properties such as *any train entering a route will eventually leave it* needs more effort, and further work is needed to efficiently verify such properties. Besides, until now the model is only designed to verify single interlockings. This assumption holds for relatively small stations such as our case study but is not true for larger stations where several interlockings communicate together. The next step will be to extend the model in order to embed the verification of a set of communicating interlockings. Finally, as we plan to verify larger interlocking systems, we will have to work on increasing the efficiency of model-checking algorithms.

References

[1] (2015): *The XML-Interface for Railway Applications - http://www.railml.org*. Available at `http://www.railml.org`.

[2] Robert Abo & Laurent Voisin (2013): *Data Formal Validation of Railway Safety-Related Systems: Implementing the OVADO Tool*. *FM-RAIL-BOK, Workshop 2013, Madrid*, pp. 27–32, doi:10.1007/978-3-319-05032-4_17.

[3] Hubert Bellon (2014): *Data Preparation Guide for Interlocking used in the Belgian Railways*. Infrabel - technical references N20.

[4] Simon Busard & Charles Pecheur (2013): *PyNuSMV: NuSMV as a Python Library*. In Guillaume Brat, Neha Rungta & Arnaud Venet, editors: Nasa Formal Methods 2013, LNCS 7871, Springer-Verlag, pp. 453–458, doi:10.1007/978-3-642-38088-4_33.

[5] CENELEC (2011): *EN50128 - Railway applications - Communication, Signalling and Processing Systems - Software for Railway Control and Protection Systems*.

[6] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani & Armando Tacchella (2002): *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. In Ed Brinksma & KimGuldstrand Larsen, editors: Computer Aided Verification, Lecture Notes in Computer Science 2404, Springer Berlin Heidelberg, pp. 359–364, doi:10.1007/3-540-45657-0_29.

[7] Edmund M. Clarke, Orna Grumberg & Doron Peled (2001): *Model checking*. MIT Press, doi:10.1016/B978-044450813-3/50026-6. Available at `http://books.google.de/books?id=Nmc4wEaLXFEC`.

[8] Alessandro Fantechi, Wan Fokkink & Angelo Morzenti (2012): *Some Trends in Formal Methods Applications to Railway Signaling*, pp. 61–84. John Wiley & Sons, Inc., doi:10.1002/9781118459898.ch4.

[9] Michael Huber & Steve King (2002): *Towards an Integrated Model Checker for Railway Signalling Data*. Springer-Verlag Berlin Heidelberg 2002, p. 20, doi:10.1007/3-540-45614-7_12.

[10] Ahmad Mirabadi & Mohammad B. Yazdi (2009): *Automatic Generation and Verification of Railway Interlocking Control Tables using FSM and NuSMV*. Transport Problems : an International Scientific Journal 4, pp. 103–110. Available at `http://www.transportproblems.polsl.pl/pl/Archiwum/2009/zeszyt1/2009t4z1_13.pdf`.

[11] Faraon Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider & Helen Treharne (2012): *Combining Event-based and State-based Modeling for Railway Verification*. Computing Sciences Report.

[12] Faraon Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider & Helen Treharne (2012): *CSP||B Modelling for Railway Verification: The Double Junction Case Study*. Proceedings of the 12th International Workshop on Automated Verification of Critical Systems.

[13] Faron Moller, HoangNga Nguyen, Markus Roggenbach, Steve Schneider & Helen Treharne (2013): *Defining and Model Checking Abstractions of Complex Railway Models Using CSP||B*. In Armin Biere, Amir Nahir & Tanja Vos, editors: Hardware and Software: Verification and Testing, Lecture Notes in Computer Science 7857, Springer Berlin Heidelberg, pp. 193–208, doi:10.1007/978-3-642-39611-3_20.

[14] George Raymond (2014): *Where are the CENELEC standards going ?* IRSE News Issue 203, pp. 21–23.

[15] Kirsten Winter & Neil J. Robinson: *Modelling Large Railway Interlockings and Model Checking Small Ones*. In: In Michael Oudshoorn, editor, Twenty-Fifth Australasian Computer Science Conference (ACSC2003), pp. 309–316.

Automatic Generation of Minimal Cut Sets

Sentot Kromodimoeljo and Peter A. Lindsay
School of IT&EE, The University of Queensland, St Lucia Qld 4072, Australia

A cut set is a collection of component failure modes that could lead to a system failure. Cut Set Analysis (CSA) is applied to critical systems to identify and rank system vulnerabilities at design time. Model checking tools have been used to automate the generation of minimal cut sets but are generally based on checking reachability of system failure states. This paper describes a new approach to CSA using a Linear Temporal Logic (LTL) model checker called BT Analyser that supports the generation of multiple counterexamples. The approach enables a broader class of system failures to be analysed, by generalising from failure state formulae to failure behaviours expressed in LTL. The traditional approach to CSA using model checking requires the model or system failure to be modified, usually by hand, to eliminate already-discovered cut sets, and the model checker to be rerun, at each step. By contrast, the new approach works incrementally and fully automatically, thereby removing the tedious and error-prone manual process and resulting in significantly reduced computation time. This in turn enables larger models to be checked. Two different strategies for using BT Analyser for CSA are presented. There is generally no single best strategy for model checking: their relative efficiency depends on the model and property being analysed. Comparative results are given for the A320 hydraulics case study in the Behavior Tree modelling language.

Keywords: Behavior Trees; minimal cut sets; model checking; safety analysis

1 Introduction

A *cut set* is a collection of component failures that could lead to a system failure. A cut set is *minimal* if none of its proper subsets are themselves cut sets. Cut Set Analysis (CSA) is the discovery of a complete set of minimal cut sets (MCSs) for given system failure modes. CSA, or an equivalent method such as Fault Tree Analysis (FTA), is typically mandated by standards for critical systems (e.g. [11]) to identify and rank system vulnerabilities at design time.

Traditionally, a system failure mode, or *top event* to give it its technical name, has to be identified before CSA can proceed. Model checkers are often used to automate CSA when the top event can be characterised by a state formula. CSA proceeds by determining if the top event is reachable for various component failure-mode combinations (the cut sets). Of particular interest are failure combinations that are minimal (the MCSs). For some modelling notations, CSA has been fully automated [2, 3, 4].

Rae and Lindsay [26] generalised the characterisation of system failure in FTA to violation of a temporal property rather than simply a state formula. This approach enables a broader class of system failures to be examined, such as state changes under circumstances when states shouldn't change, and action sequences occurring in an undesirable order. While such properties can sometimes be captured by adding an observer automaton to the model, it is often more natural to state the property as a temporal property, and desirable not to modify the model [16]. And in other examples the failures themselves are actually behaviors rather than events or conditions: for example Cerone *et al* [6] classify human failures by repeated patterns of behaviour, in a highly interleaved cognitive task where it is impossible to say exactly when the failure occurred.

Lindsay *et al* [17] developed a semi-automated approach to CSA with this richer notion of failures, using Behavior Trees (BT) to model the system with potential component failure-behaviours injected.

J. Pang, Y. Liu, and S. Mauw (Eds.): 4th International Workshop
on Engineering Safety and Security Systems 2015 (ESSS'15)
EPTCS 184, 2015, pp. 33–47, doi:10.4204/EPTCS.184.3

The SAL model checker [20] was used iteratively to identify MCSs, with the temporal property reformulated manually at each step to ignore previously discovered MCSs. The manual steps were tedious and error-prone. Moreover, in order to generate a complete CSA, the model structure needed to be "flat", in the sense that if failure event $C1$ is preceded by failure event $C2$ in some path, then there also exists a path in which $C1$ occurs but $C2$ does not. (This is a stronger assumption than simply that failures are independent.)

This paper describes a completely new approach to automation of CSA, using a new model checker called BT Analyser, in which search for counterexamples can be directed [13]. As well as avoiding the need for manual steps and the assumption about non-dependence of component failures, the checker is significantly more efficient than the one used in [17], with capabilities that go well beyond CSA. The automation takes advantage of some novel features of BT Analyser including the use of *cycle constraints* for counterexample generation, *global constraints*, and incremental analysis, resulting in an efficient tool for CSA. This paper also compares alternative strategies for the automatic generation of minimal cut sets to illustrate the virtues of the novel features of BT Analyser. The approach is illustrated on a BT model to enable comparison with previous approaches [17] but it is applicable to any modelling notation that can be translated to a finite state transition system.

- Section 2 describes the modelling framework and introduces CSA terminology and concepts.
- Section 3 describes the LTL model checker for Behavior Trees.
- Section 4 describes the techniques for generating minimal cut sets.
  - Section 4.1 describes how a cut set is extracted from a counterexample path.
  - Section 4.2 describes alternative approaches for verifying the minimality of a cut set.
  - Section 4.3 discusses two strategies for automating the generation of all minimal cut sets.
- Section 5 presents results of experiments on applying the techniques to the Airbus A320 hydraulic system case study used in [3] and [17].
- Section 6 discusses application of the approach to other modelling notations.
- Sections 7, 8 and 9 discusses related work, provides a summary and conclusion, and discusses future work, respectively.

2 Terminology and assumptions

The core of BT Analyser works for a very general class of modelling languages, including asynchronous finite-state systems with interleaving semantics [13]. It does so by using a modelling framework based on typed multi-variable state transition systems in which the effect of each transition is deterministic (but the choice of transition is non-deterministic) and the truth value of each atomic proposition in a state is determinable. Modelling languages that can be translated into this framework include state machines, Behavior Trees and activity diagrams.

In what follows we assume the component failure modes of interest have already been injected into the model of system behaviours: that is, the analyst has determined all of the component failure modes that are going to be analysed and has included events (called *basic events* in CSA and FTA) corresponding to occurrences of component failures. The effects of a component failure may need to be reflected in the system behaviour. However, we do not assume that a failed component remains failed. As usual for causal analysis techniques such as CSA and FTA, we assume component failures are independent: common mode failures are analysed after CSA [14].

For the purpose of CSA, we assume that basic events are encoded using special Boolean state variables, called *basic event flags* below. Each of these state variables represents the occurrence of the corresponding event. Initially, the state variable must have the value *false*. When the corresponding event occurs, the value of the state variable must change to *true* and no behaviour can change the value back to *false*. If the analyst's modelling notation does not support this, it can be easily added during translation to BT Analyser's modelling framework. Note that it is the event that cannot be undone, not the failure of a component.

Given a system safety property *Safe* expressed in Linear Temporal Logic (LTL) [25], the model checker investigates whether there is a *counterexample* to *Safe*: that is, a (possibly infinite) sequence π of transitions through the model which satisfies $\neg$*Safe*. (In fact the model checker looks for paths with a finite prefix p and an infinitely repeated subpath c, but if there is a counterexample at all then there is always one in the above form.) Note that *safety property* in this paper means the formalisation of a system safety requirement, rather than the narrow technical sense of $\mathbf{G}\neg P$ for a state condition P.

With the top event replaced by "a violation of system safety property", a cut set is the set of basic events that occur in a behaviour that violates the system safety property (a counterexample path for *Safe*). The set of basic events that occur in a counterexample path can be extracted from the cycle part of the counterexample path: they are exactly the basic events whose corresponding basic event flags have the value *true* in the cycle (recall that basic event flags can only change value from *false* to *true*, so in a cycle they must maintain their values).

Two different strategies are given in Section 4 below for generating counterexamples corresponding to MCSs.

3 BT Analyser: A Symbolic LTL Model Checker

BT Analyser is a symbolic LTL model checker for Behavior Trees. It implements novel techniques for LTL model checking and counterexample generation described in [13]. Although BT Analyser only accepts the Behavior Tree (BT) notation as the modelling notation, the core of the model checker is notation-independent.

Novel features of BT Analyser include:

- Directed counterexample path generation with cycle constraints and global constraints.
- Computation of fair states with global constraints.
- On-the-fly symbolic LTL model checking as well as the traditional fixpoint approach to symbolic model checking.
- Incremental analysis.

BT Analyser is a symbolic model checker, operating on sets of states characterised using propositions rather than individual states. A *symbolic state* is a proposition characterising a set of states.

BT Analyser's framework uses *elementary blocks* as abstract transitions. The effect of an elementary block transition is deterministic; the non-determinism is in choosing an elementary block to transition, when several are enabled.

For each elementary block b, let f_b be the function that computes the symbolic image under the transition (sub-)relation for b and let r_b compute the symbolic preimage. If B is the set of elementary blocks for the system modelled, then the function f_B for computing the symbolic image under the transition relation for the system is defined as

$$f_B(S) \triangleq \bigvee_{b \in B} f_b(S).$$

Similarly for the preimage,

$$r_B(S) \triangleq \bigvee_{b \in B} r_b(S).$$

Propositional operations are performed using ordered binary decision diagrams (OBDDs) [5].

Following model checking convention, let the negation of the system safety behaviour being analysed be denoted φ. During model checking, a *tableau* (decision graph) for φ is superimposed on the model producing an augmented model (see [13]). The image and preimage functions for the augmented model are denoted f'_b and r'_b for an elementary block b and f'_B and r'_B for the entire system.

Definition 1 (Symbolic Counterexample Path)**.** A symbolic counterexample path has the form of a triple:

$$(I_\pi, p_\pi, s_\pi)$$

where I_π characterises a set of initial states, prefix p_π is a possibly empty finite sequence of elementary blocks, and cycle s_π is a non-empty finite sequences of elementary blocks that is repeated forever. A symbolic counterexample path represents a set of paths in the model that satisfy φ (the paths share the abstract transitions and all satisfy φ).

The intermediate symbolic states in a symbolic path can be computed from I_π and the image functions for the transitions (the elementary blocks) in the symbolic path.

Directed counterexample path generation allows BT Analyser to be told to find a symbolic counterexample path in which a cycle constraint cc is satisfied by a symbolic state in the cycle part of the path, and a global constraint gc is satisfied by all symbolic states in the path.

In the fixpoint approach to symbolic LTL model checking, the set of augmented states (states in the augmented model) that are fair with respect to φ is computed. A set of fairness constraints C_φ is defined in a manner that is dependent on the LTL encoding scheme. The fixpoint characterisation of the set of augmented states that are fair with respect to φ is

$$F_\varphi \triangleq \nu Z. \bigwedge_{c \in C_\varphi} r'_B(\mu Y.(Z \wedge c) \vee r'_B(Y))$$

where ν and μ are respectively the greatest fixpoint and least fixpoint operators of μ-calculus [12]. The inner least fixpoint expression characterises states that satisfy $Z \wedge c$ directly or do not satisfy $Z \wedge c$ but can reach states that satisfy $Z \wedge c$ in the augmented model, where Z is the variable of the outer greatest fixpoint operation. The outer greatest fixpoint operation interacts with the inner fixpoint operations, combining to ensure that each fairness constraint $c \in C_\varphi$ is satisfied infinitely often in each path whose states are in F_φ. (The overall fixpoint expression characterises states that can transition to states that can start paths in which each $c \in C_\varphi$ is satisfied infinitely often.)

A symbolic augmented state S_φ, defined according to the LTL encoding scheme, characterises the set of augmented states that are "commited" to starting paths that satisfy φ. Let I_φ denote the symbolic augmented state characterising the set of initial augmented states. The intersection characterised by

$$I_\varphi \wedge S_\varphi \wedge F_\varphi$$

determines if there is a counterexample for the LTL specification: the intersection is empty if and only if the LTL specification has no counterexamples.

BT Analyser allows the computation of augmented states that are fair with respect to φ while satisfying a global constraint gc globally. Note that in general this is not equivalent to $F_\varphi \wedge gc$. Instead, the following fixpoint characterisation can be used:

$$\nu Z. \quad gc \wedge \bigwedge_{c \in C_\varphi} r'_B(\mu Y.(Z \wedge c) \vee r'_B(Y)). \tag{1}$$

The intersection of the set of augmented states characterised by (1) with the set characterised by $I_\varphi \wedge S_\varphi$ determines if there is a counterexample path for the LTL specification for which *gc* holds globally.

In addition to the fixpoint approach to symbolic model checking, the model checker also allows on-the-fly symbolic LTL model checking. The algorithm used is an adaptation of the standard nested DFS algorithm [9] with the following important differences:

- The adapted algorithm works with symbolic transitions (using elementary blocks) and symbolic states.
- The adapted algorithm constructs the Büchi automaton implicitly and on-the-fly.

Details of the algorithm can be found in [13].

Finally, an important feature of BT Analyser is the ability to perform analysis incrementally. As an example, an analysis can start with reachability analysis, followed by computation of fair states, followed by computation of counterexample states (states that can be part of counterexample paths), followed by the generation of a counterexample path. We can then tell BT Analyser to find more counterexample paths as well as fair states with global constraints without redoing the initial parts of the analysis. This enables different strategies to be used for different problems. An example is provided in [13] where reachability analysis before model checking is a good strategy for a prioritised BT model (where internal system transitions are prioritised over external events) but is a bad strategy for a non-prioritised BT model.

4 Generating Minimal Cut Sets

We assume that component failure modes have been injected into the model. An LTL specification whose violation represents a system failure is first model checked. If a traditional top event represented by a state formula *top* is to be used, then the LTL specification model checked would be $\mathbf{G}\ \neg top$. Using our method, $\mathbf{G}\ \neg top$ can be replaced by any LTL formula representing non-occurrence of the behaviour associated with system failure.

In what follows let φ denote the hazardous system behaviour being analysed (i.e., the negation of the LTL formula representing the desired system safety property).

Cut sets will be extracted from counterexample paths and verified to be minimal. The entire process of generating all MCSs can be fully automated.

4.1 Extracting Cut Sets from Counterexample Paths

Traditionally, a cut set is the set of basic events that occur in a behaviour leading to the top event. Unfortunately, LTL does not include events. Even the more expressive temporal logic CTL* (see e.g., [8]) does not include events. The occurrence of an event is obviously irreversible: once an event occurs, from that point on, the event has occurred. A simple way to encode the occurrence of an event is to use a Boolean state variable, say *eOccurred*. Initially the value of *eOccurred* is *false*. When the event occurs, *eOccurred* is set to *true* and no behaviour can change it back to *false*. This way of encoding the occurrence of an event is used by Lindsay *et al* in their method [17]. We call the state variable for a basic event a *basic event flag*. A cut set can be encoded as a conjunction of the basic event flags that correspond to elements of the cut set.

We can extract the cut set by choosing a symbolic state in the cycle part and pick all positive occurrences of basic event flags. As an example, suppose there are 11 basic components that can fail and their

failure occurrences are represented by the basic event flags

$$distyF, distgF, distbF, E1F, E2F, PTUF, EDPyF, EDPgF, EMPbF, EMPyF, \text{ and } RATF.$$

We will denote the number of basic events MAX. Thus for the above example $MAX = 11$. Suppose further that we choose the following symbolic state from the cycle part of the symbolic counterexample path (taken from an actual run of the model checker on the A320 hydraulics case study):

$$\begin{array}{ll} & (Pilot = ready) \wedge (Engine1 = on) \wedge (Engine2 = off) \wedge (Yellow = off) \\ \wedge & (PTUy = off) \wedge (PTU = off) \wedge (EDPy = off) \wedge (EDPg = on) \wedge (Blue = off) \\ \wedge & (EMPy = off) \wedge (EMPb = off) \wedge (RAT = off) \wedge \neg E1F \wedge E2F \wedge \neg distyF \\ \wedge & \neg distgF \wedge \neg distbF \wedge PTUF \wedge \neg EDPyF \wedge \neg EDPgF \wedge EMPbF \wedge EMPyF \\ \wedge & \neg RATF \wedge (Green = on) \wedge (Aircraft = flyingSlow) \wedge (System = operating), \end{array}$$

then the cut set extracted from the counterexample path would be

$$\{E2F, PTUF, EMPbF, EMPyF\}.$$

Any symbolic state in the cycle can be chosen, thus we can always choose the starting symbolic state of the cycle (basic event flags can only transition from *false* to *true*, thus their values must remain constant throughout a cycle).

If the symbolic state has a disjunction (i.e., the symbol "$\vee$" occurs in the proposition that is the symbolic state), then the symbolic state can be normalised into an irredundant sum of product (ISOP) form and a cut set can be extracted from any of the disjuncts. In BT Analyser, ISOP forms are generated directly from OBDDs using the Minato-Morreale algorithm [18, 19].

4.2 Verifying Minimality of Cut Sets

In isolation, if a cut set is extracted from a counterexample path, then it can be checked for minimality by ensuring that there are no counterexample paths if any member of the cut set is "removed". This can be performed in the model checker by computing the set of states that are fair with respect to φ with a global constraint gc representing the removal of a member of the cut set. The computation is performed using (1). If the intersection of the resulting set of states with the set of initial states is empty, then the cut set is minimal. The idea goes as follows. Continuing with the example from Section 4.1, suppose the cut set of interest (denoted cs) is

$$cs = \{E2F, PTUF, EMPbF, EMPyF\}.$$

The gc is a conjunction of all the other components not failing together with at least one of the members of the cut set also not failing, and for the above cs we would have

$$\begin{array}{rcl} gc & \equiv & \neg distyF \wedge \neg distgF \wedge \neg distbF \wedge \neg E1F \wedge \neg EDPyF \wedge \neg EDPgF \wedge \neg RATF \\ & & \wedge (\neg E2F \vee \neg PTUF \vee \neg EMPbF \vee \neg EMPyF). \end{array}$$

The conjunction $\neg distyF \wedge \neg distgF \wedge \neg distbF \wedge \neg E1F \wedge \neg EDPyF \wedge \neg EDPgF \wedge \neg RATF$ represents components that have not failed and $(\neg E2F \vee \neg PTUF \vee \neg EMPbF \vee \neg EMPyF)$ represents the removal of at least one component from cs. If the intersection of the result of (1) with the set of initial states is not empty, then there is a counterexample path with global constraint gc, meaning there is a proper subset of cs that is a cut set, and thus cs is non-minimal. Otherwise cs is minimal.

Computing the set of states that are fair with respect to φ with global constraint *gc* is usually much faster than computing the set of states that are fair with respect to φ without global constraints. However, often this is still more expensive than counterexample path generation. With some strategies to be discussed in Section 4.3, minimality need not be checked and the number of computations of fair states can be reduced.

4.3 Strategies for Generating Minimal Cut Sets

Two different strategies for automating CSA are given here: a naive strategy and a systematic strategy. The naive strategy finds MCSs in no specific order, while the systematic strategy finds MCSs in a non-decreasing order in terms of size.

4.3.1 Naive Strategy

The steps of this strategy are as follows:

1. Compute the set of fair states.
2. Set $gc \leftarrow true$ (i.e., no global constraints).
3. Find a counterexample path with global constraint *gc* and no cycle constraints.
4. If a counterexample path is found, extract a cut set and verify its minimality as described in Section 4.2. If it is not minimal then find a counterexample path for the minimality and extract a smaller cut set. This can be repeated until a minimal cut set is extracted. Add the MCS to the set of MCSs sets found and modify *gc* to rule out the found MCS from further consideration: add the negation of a representation of the MCS as a conjunct to *gc*. For the above example cut set, the negation is

   $$\neg(E2F \wedge PTUF \wedge EMPbF \wedge EMPyF).$$

   Repeat from step 3.
5. Otherwise no counterexample path was found that satisfies *gc* globally and all minimal cut sets have been found.

A proof sketch of the correctness of the strategy is as follows:

- Each conjunct in *gc* is associated with an MCS already found and rules out any more cut sets that are subsumed by the MCS.
- In step 4, each time an MCS is found, an appropriate conjunct is added to *gc*.
- Nowhere else is a conjunct added to *gc*.
- If there are no counterexamples that satisfy *gc* then there are no cut sets except those that are subsumed by the MCSs found in step 4 (since *gc* rules out cut sets that are subsumed by the MCSs already found).

Since after step 5 we can conclude that all cut sets are subsumed by the MCSs found, the MCSs found are exactly all the MCSs.

4.3.2 Systematic Strategy

As mentioned in Section 4.2, verifying the minimality of a cut set by computing fair states with global constraints can be expensive. A more systematic strategy that could be more efficient would be to find all MCSs of size 0, then all MCSs of size 1, then all MCSs of size 2, and so on until we reach size *MAX*. This strategy uses the cycle constraint feature of directed counterexample generation:

1. Compute the set of fair states.
2. Set $gc \leftarrow true$ and $n \leftarrow 0$.
3. Find a counterexample path with global constraint *gc* and cycle constraint "there are *n* or less basic events". For example, suppose there are 3 basic events represented by $C1F, C2F$ and $C3F$. For $n = 1$ the cycle constraint would be

   $$\neg C1F \wedge \neg C2F \vee \neg C1F \wedge \neg C3F \vee \neg C2F \wedge \neg C3F.$$

   It may be easier to view the constraint as "at least $MAX - n$ basic events have not happened". It is equivalent to the standard "choosing $MAX - n$ out of *MAX*" combinatorial problem.
4. If a counterexample path is found, the extracted cut set would be a minimal cut set. Add the MCS to the set of MCSs found and modify *gc* to rule out the found MCS from further consideration (as in step 4 of the naive strategy). Repeat from step 3.
5. Otherwise no counterexample path was found that satisfies the constraint. If $n < MAX$, set $n \leftarrow n + 1$ and repeat from step 3. Otherwise $n \geq MAX$, and all MCSs have been found.

A proof sketch of the correctness of the strategy is as follows (it is slightly different than the one for the naive strategy):

- Each conjunct in *gc* is associated with an MCS already found and rules out any more cut sets that are subsumed by the MCS.
- Each time an MCS is found in step 3, an appropriate conjunct to rule out the MCS found from further consideration is added to *gc* in step 4.
- Nowhere else is a conjunct added to *gc*.
- If no more counterexample is found in step 3, then there are no cut sets of size $\leq n$ except those subsumed by the MCSs already found.
- A cut set found in step 3 is a minimal cut set: the case where $n = 0$ is trivial; for $1 \leq n \leq MAX$, we can assume that all cut sets of size $\leq n - 1$ are subsumed by the MCSs already found therefore any cut set found in step 3 is a minimal cut set (since they are not subsumed by the MCSs already found, thus they are not subsumed by any cut set of size $< n$).
- If there are no counterexamples that satisfy *gc* when $n = MAX$ then there are no cut sets of size $\leq MAX$ except those that are subsumed by the MCSs already found.

Since the size of a cut set cannot be greater than *MAX*, after step 5 with $n = MAX$ we can conclude that all cut sets are subsumed by the MCSs found, thus the MCSs found are exactly all the MCSs.

5 Experimental Results

Experiments were conducted to compare the performances of the two strategies described in Section 4.3. The naive strategy was then slightly modified, with on-the-fly symbolic LTL model checking replacing directed counterexample path generation, and the performance of the modified strategy compared with that of the original strategy.

Table 1: A320 Hydraulics Timing Results

Strategy	Initial MC run	MCS Generation	Total
Naive	8 minutes	57 minutes	65 minutes
Systematic	8 minutes	5 minutes	13 minutes

5.1 Comparison of Two Strategies

The two strategies described in Section 4.3 were applied to the A320 hydraulics case study used in [3] and [17]. The BT model with the injected faults is included as an example for BT Analyser, which can be accessed using the DOI 10.14264/uql.2015.16 (it is part of the supplementary material for [13]). The experiments were performed on a notebook computer with a 2.7GHz i7 CPU and 16GB of RAM running Ubuntu. The BT model is similar to the one used in [17]. The LTL specification used is

$$\mathbf{G}\left(\neg \mathit{INIT} \vee \mathbf{G}\left(\begin{array}{ll} & (\mathit{Yellow}=\mathit{on}) \wedge (\mathit{Green}=\mathit{on}) \\ \vee & (\mathit{Yellow}=\mathit{on}) \wedge (\mathit{Blue}=\mathit{on}) \\ \vee & (\mathit{Green}=\mathit{on}) \wedge (\mathit{Blue}=\mathit{on}) \end{array}\right)\right)$$

where **G** is the *globally* temporal operator ($\Box$ or box), and

$$\mathit{INIT} \triangleq (\mathit{Yellow}=\mathit{on}) \wedge (\mathit{Green}=\mathit{on}) \wedge (\mathit{Blue}=\mathit{on}).$$

Both strategies produced the same set of MCSs, although the order in which the MCSs were found were different. Five 2-element MCSs were found:

$$\{\mathit{distyF}, \mathit{distbF}\}, \{\mathit{distyF}, \mathit{distgF}\}, \{\mathit{distgF}, \mathit{distbF}\}, \{\mathit{distyF}, \mathit{EMPbF}\} \text{ and } \{\mathit{distgF}, \mathit{EMPbF}\}.$$

There were ten 3-element MCSs found:

$$\{\mathit{distyF}, \mathit{PTUF}, \mathit{EDPgF}\}, \{\mathit{distbF}, \mathit{PTUF}, \mathit{EDPgF}\}, \{\mathit{distyF}, \mathit{E1F}, \mathit{PTUF}\},$$
$$\{\mathit{distbF}, \mathit{E1F}, \mathit{PTUF}\}, \{\mathit{PTUF}, \mathit{EDPgF}, \mathit{EMPbF}\}, \{\mathit{E1F}, \mathit{PTUF}, \mathit{EMPbF}\},$$
$$\{\mathit{EDPyF}, \mathit{EDPgF}, \mathit{EMPyF}\}, \{\mathit{E1F}, \mathit{E2F}, \mathit{EMPyF}\}, \{\mathit{E2F}, \mathit{EDPgF}, \mathit{EMPyF}\}$$
$$\text{and } \{\mathit{E1F}, \mathit{EDPyF}, \mathit{EMPyF}\}.$$

There were six 4-element MCSs found:

$$\{\mathit{distbF}, \mathit{PTUF}, \mathit{EDPyF}, \mathit{EMPyF}\}, \{\mathit{distgF}, \mathit{PTUF}, \mathit{EDPyF}, \mathit{EMPyF}\},$$
$$\{\mathit{distbF}, \mathit{E2F}, \mathit{PTUF}, \mathit{EMPyF}\}, \{\mathit{distgF}, \mathit{E2F}, \mathit{PTUF}, \mathit{EMPyF}\},$$
$$\{\mathit{PTUF}, \mathit{EDPyF}, \mathit{EMPbF}, \mathit{EMPyF}\} \text{ and } \{\mathit{E2F}, \mathit{PTUF}, \mathit{EMPbF}, \mathit{EMPyF}\}.$$

Table 1 shows the overall timing results for the two strategies. Both strategies have the same initial model checking time for the LTL specification which is 8 minutes. The subsequent MCS generation times differed substantially.

The use of cycle constraints drastically reduced the counterexample generation time. As Fig. 1 shows, the counterexample generation times for the naive strategy started at a high of 490 seconds and dropped significantly as more global constraints were added. In contrast, the counterexample generation times for the systematic strategy (which uses cycle constraints) did not vary as much and were quick at between 3 and 9 seconds (see Fig. 2).

The times for computing fair states with global constraints to check minimality did not vary so much, as can be seen in Fig. 3. They were between 40 and 60 seconds. In fact, the four cases where the times

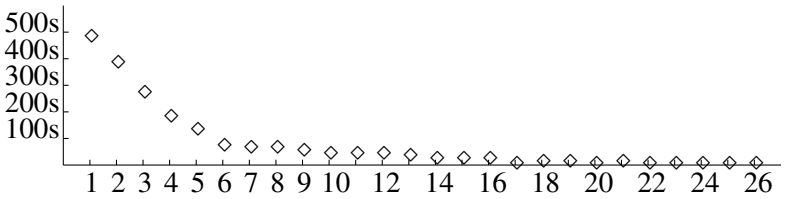


Figure 1: CTR Generation Time for Naive Strategy (26 counterexample paths)

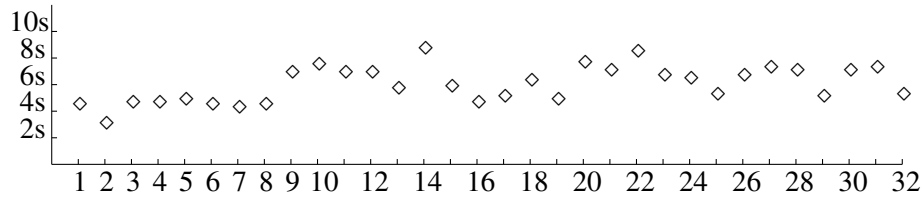


Figure 2: CTR Generation Time for Systematic Strategy (32 counterexample paths)

were significantly higher were exactly for cases where the cut set turned out to be non-minimal: cases 16, 19, 21 and 24. Interestingly, the performance penalty for those cases were somewhat offset by the significantly faster counterexample generation times that immediately followed: cases 17, 20, 22 and 25 in Fig. 1. There is no case 26 for checking minimality since no counterexample was found for the case.

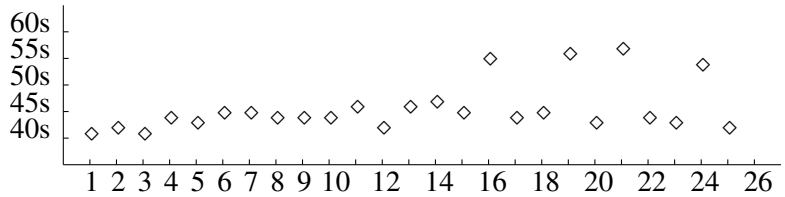


Figure 3: Minimality Checking Time for Naive Strategy (25 iterations)

The results from the experiments clearly show that cycle constraints in directed counterexample generation can drastically reduce the computation time required to find counterexample paths. Moreover, cycle constraints can be used to direct the counterexample path generation towards specific kinds of paths. For generating MCSs using the systematic strategy, the cycle constraint directs the model checker to find a counterexample path that produces a cut set of exactly the desired size.

The results also show the utility of global constraints. Global constraints serve two purposes:

- to rule out certain classes of counterexamples, and
- to reduce the search space for counterexamples.

The second is a pleasant consequence of the first. Generally, a smaller search space results in faster search time. The effect is rather dramatic in counterexample generation for the naive strategy (Fig. 1).

Finally, the results show the benefits of incremental analysis described in Section 3. Even a bad strategy (the naive strategy) is helped by incremental analysis. The computation time using the bad strategy is still better than the total computation time, using SAL (rerun on the same notebook computer used for the experiments), of the original approach in [17] (1 hour vs 4 hours). Although there are too many factors that are not taken into account for a fair comparison, it is clear that not redoing work can save a lot of time.

Table 2: Results for Naive Strategy with Symbolic On-the-fly LTL Checking

$\|ICS\|$	$\|MCS\|$	CTR Existence Test	Total OTF	Total Minimality Test
4	3	415.5s	6.8s	141s
6	3	414.0s	21.4s	246.7s
9	2	321.1s	72.2s	833.0s
8	2	271.8s	39.8s	738.1s
9	3	239.8s	81.8s	812.8s
8	2	346.7s	38.6s	781.6s
5	3	262.5s	23.3s	194.1s
5	2	254.3s	25.5s	361.7s
5	3	244.3s	23.5s	266.0s
7	2	230.6s	33.4s	615.0s
5	3	206.3s	24.6s	272.6s
5	3	210.2s	27.5s	196.9s
5	3	201.1s	27.9s	268.8s
6	3	191.6s	28.9s	385.9s
5	4	192.8s	22.2s	166.4s
6	4	190.0s	30.4s	279.2s
6	4	193.1s	28.2s	282.1s
4	3	191.2s	28.9s	128.6s
4	4	169.3s	26.4s	57.8s
5	4	164.7s	31.3s	167.5s
5	4	164.6s	33.4s	164.3s
		98.1s		
		5173.6s	676.0s	7360.1s

5.2 Using On-the-fly Symbolic LTL Model Checking

Recall from Section 3 that on-the-fly symbolic LTL model checking can be used to find counterexample paths quickly. However, the on-the-fly symbolic LTL model checking facility in BT Analyser is not as well developed as directed counterexample path generation. In particular the facility does not yet support cycle constraints. As a result, the facility cannot be used for the systematic strategy.

An experiment was conducted with on-the-fly symbolic LTL model checking replacing directed counterexample path generation for the naive strategy. In the experiment, the existence of any remaining counterexample path (and thus any remaining MCSs) was checked first in each round of MCS generation. This is because on-the-fly symbolic LTL checking might take a long time if there are no counterexamples.

Table 2 shows the results of the experiment with on-the-fly symbolic LTL checking. For each round (represented by a row), the column $|ICS|$ is for the size of the initial cut set found, the column $|MCS|$ is for the size of the MCS found, the column "CTR Existence Test" is for the time it took to check the existence of a counterexample path (by computing fair states), the column "Total OTF" is for the total of the on-the-fly symbolic LTL checking times, and the column "Total Minimality Test" is for the total of the minimality test times. The number of iterations for a round is $|ICS| - |MCS| + 1$, thus for $|ICS| = 9$ and $|MCS| = 2$, there would have been 8 iterations of on-the-fly symbolic LTL checking and minimality test in the round. The second last row represents the time needed to verify that there are no more MCSs.

The results show that, although the total of on-the-fly symbolic LTL checking times (11 minutes) is much less than the total of the counterexample generation times for the naive strategy with directed counterexample path generation (57 minutes), the overall analysis time is significantly greater. Even if we ignore the existence test times for the rounds and use 98.1 seconds as the time to check the absence of more counterexamples (we can imagine a parallelised implementation where on-the-fly symbolic LTL checking is run in parallel with checking the absence of counterexamples and whichever result is obtained first is used, with the other computation aborted), the total time is still around 135 minutes, compared to 65 minutes using directed counterexample path generation.

The disappointing performance when using on-the-fly symbolic LTL checking for the naive strategy appears to be caused by the *depth-first-search with imprecise (but safe) tracking of fairness constraints* nature of the algorithm used. This resulted in counterexample paths that tended to produce cut sets with many more failed components than necessary. Of the 21 MCSs, only one of them was found in one iteration, with the other 20 MCSs requiring 78 iterations in all. (By contrast, with directed counterexample path generation, 17 of the MCSs were found in one iteration and the remaining 4 were found in two iterations.) Thus, any savings from the fast counterexample generation times were completely wiped out by the times required for the minimality tests.

Perhaps if cycle constraints can be incorporated into on-the-fly symbolic LTL checking, then the systematic strategy might benefit from fast counterexample path generation times of on-the-fly symbolic LTL checking without the initial overhead of computing fair states.

6 Discussion

The approach described above, and BT Analyser, were developed in a very general typed multi-variable transition system modelling framework. The results were illustrated on a translation from the Behavior Tree notation into the framework but could easily be extended to support other modelling notations, including state-based notations such as VDM, Z and B. The main restriction is that all sets must be bounded finite sets and the number of threads is bounded. Note that if the model has a terminating state, then the state must be replaced by one that has a "null" transition (i.e., it can transition to itself). This ensures that all paths are infinite.

Fault injection in BT models is usually very straightforward. Although they are all in some sense equivalent, the BT model of a system can have different tree shapes depending on the order in which the subtrees corresponding to different requirements were integrated (see [10] for a description of the BT model development process). If they have been developed so that individual component behaviours are grouped together by component, which is very often the case, then fault injection is simply a matter of grafting a behaviour tree corresponding to the component failure mode into the tree, with the failure (basic) event at the graft point.

Failure Modes and Effects Analysis (FMEA) is a commonly used consequence analysis technique for critical systems. It consists of checking whether individual component failures can have hazardous effects on the system. Tool support can be provided for FMEA by injecting individual faults into system models and using model checking to see if undesirable system states are reachable. Grunske *et al* [15] reports experience automating FMEA using BT models and the SAL model checker. Although in principle this approach could be extended to CSA simply by injecting multiple faults into the model, one for each element of a potential cut set, in practice the limits of model checking are quickly reached due to the state explosion problem. By contrast, BT Analyser can handle system models like those in [15] in which all of the faults are injected, and it does so very efficiently.

The generalisation of *top event* from a failure state to a behaviour allows the approach to be applied to a broad class of modelling notations, including notations that do not explicitly represent state changes, such as the BT notation.

7 Related Work

Bieber *et al* [3] combine CSA and model checking in safety analysis of the A320 hydraulics case study. They use two different models, one of the static, physical architecture and an LTL model of the activation and failure behaviour of the system, and need reasoning to combine the results to prove the safety requirements, whereas our approach uses a single model and the entire analysis is automated.

Akerlund *et al* [2] propose that reliability analysis such as FTA and verification of safety requirements be performed on the same model. Their generation of MCSs uses the reachability analysis capability of a model checker, but is based on a state condition as the top event. The FSAP/NuSMV-SA safety analysis platform [4] and the DCCA method [21] also use the reachability analysis capabilities of model checkers to generate MCSs based on a state condition. In contrast, our approach is applicable when the top event is a behaviour.

Abdulla *et al* [1] use a SAT-based model checker to find MCSs for Scade models, in which component failure modes are modelled by replacing nominal flows by extended flows. Again, as with the approaches mentioned in the previous paragraph, they perform reachability analysis to a state condition.

Papadopoulos and Maruhn [23] construct a fault tree from the design model. Ortmeier and Schellhorn [22] model the fault trees themselves and verify their correctness and completeness against Statechart models. Cha *et al* [7] use the model checker UPPAAL to verify the correctness of fault trees against timed-automata models. In contrast, our approach does not need to involve fault trees.

Lindsay *et al* [17] use a model checker to generate MCSs from failure behaviour. However, some of the steps still need to be performed manually.

8 Summary and Conclusions

We have presented an approach that can be used to generate minimal cut sets automatically from a fault-injected model and a general safety requirement. The approach takes advantage of novel features of a new model checker, BT Analyser. In particular, the approach uses the incremental analysis capability of BT Analyser, directed counterexample path generation, cycle constraints and global constraints. The use of these features significantly reduces the computation time required when compared to the traditional way of using model checkers.

Experiments on an existing case study in the Behavior Tree notation show the effectiveness of the approach and the novel features of BT Analyser. The effectiveness of cycle constraints in producing counterexample paths with desired properties and in producing them quickly was a pleasant surprise. The use of constraints appears to help mitigate the state explosion problem in model checking.

9 Future Work

The on-the-fly symbolic LTL model checking facility in BT Analyser is not as developed as directed counterexample path generation. In particular, cycle constraints cannot be specified for counterexample

paths. A possible future work would be to incorporate cycle constraints into the on-the-fly symbolic LTL checking mechanism.

Another possible future work is to incorporate partial order reduction techniques [24] into on-the-fly symbolic LTL model checking in BT Analyser. This may speed-up on-the-fly checking in cases where there are no counterexamples.

10 Acknowledgements

This work was supported by Linkage Project grants LP0989363 and LP130100201 from the Australian Research Council, Raytheon Australia and Thales Australia.

References

[1] P. Abdulla, J. Deneux, G. Stålmarck, H. Ågren & O. Åkerlund (2006): *Designing Safe, Reliable Systems Using SCADE*. In: *Leveraging Applications of Formal Methods*, LNCS 4313, Springer, pp. 115–129, doi:10.1007/11925040_8.

[2] O. Akerlund, S. Nadjm-Tehrani & G. Stålmarck (1999): *Integration of Formal Methods into System Safety and Reliability Analysis*. In: *Proceedings of 17th International Systems Safety Conference*, pp. 326–336.

[3] P. Bieber, C. Castel & C. Seguin (2002): *Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System*. In: *Dependable Computing EDCC-4*, Springer, pp. 19–31, doi:10.1007/3-540-36080-8_3.

[4] M. Bozzano & A. Villafiorita (2007): *The FSAP/NuSMV-SA Safety Analysis Platform*. *International Journal on Software Tools for Technology Transfer* 9(1), pp. 5–24, doi:10.1007/s10009-006-0001-2.

[5] R.E. Bryant (1986): *Graph-Based Algorithms for Boolean Function Manipulation*. *IEEE Transactions on Computers* C-35(8), pp. 677–691, doi:10.1109/TC.1986.1676819.

[6] A. Cerone, S. Connelly & P. A. Lindsay (2008): *Formal analysis of human operator behavioural patterns in interactive surveillance systems*. *Software and Systems Modeling* 7(3), pp. 273–286, doi:10.1007/s10270-007-0072-x.

[7] S. Cha, H. Son, J. Yoo, E. Jee & P.H. Seong (2003): *Systematic Evaluation of Fault Trees using Real-time Model Checker UPPAAL*. *Reliability Engineering & System Safety* 82(1), pp. 11 – 20, doi:10.1016/S0951-8320(03)00059-0.

[8] E.M. Clarke, Jr., O. Grumberg & D.A. Peled (1999): *Model Checking*. MIT Press.

[9] C. Courcoubetis, M.Y. Vardi, P. Wolper & M. Yannakakis (1992): *Memory-Efficient Algorithms for the Verification of Temporal Properties*. *Formal Methods in System Design* 1(2/3), pp. 275–288, doi:10.1007/BF00121128.

[10] R.G. Dromey (2003): *From Requirements to Design: Formalizing the Key Steps*. In: *1st International Conference on Software Engineering and Formal Methods*, IEEE Computer Society, pp. 2–11, doi:10.1109/SEFM.2003.1236202.

[11] International Electrotechnical Commission (2010): *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. Part 1: General requirements*. International Standard IEC 61508-1.

[12] D. Kozen (1983): *Results on the Propositional mu-Calculus*. *Theoretical Computer Science* 27, pp. 333–354, doi:10.1016/0304-3975(82)90125-6.

[13] S. Kromodimoeljo (2014): *Controlling the Generation of Multiple Counterexamples in LTL Model Checking*. phdthesis, doi:10.14264/uql.2015.16.

[14] N. Leveson (1995): *Safeware - System Safety and Computers: A Guide to Preventing Accidents and Losses caused by Technology*. Addison-Wesley.

[15] L.Grunske, K. Winter, N. Yatapanage, S. Zafar,Saad & P.A. Lindsay (2011): *Experience with Fault Injection Experiments for FMEA*. *Software: Practice and Experience* 41(11), pp. 1233–1258, doi:10.1002/spe.1039.

[16] P.A. Lindsay, K. Winter & S. Kromodimoeljo (2012): *Model-based Safety Risk Assessment using Behavior Trees*. In: *Proceedings of the 6th Asia Pacific Conference on System Engineering*, Systems Engineering Society of Australia. Available at `http://staff.itee.uq.edu.au/pal/papers/SETE2012.pdf`.

[17] P.A. Lindsay, N. Yatapanage & K. Winter (2012): *Cut Set Analysis using Behavior Trees and Model Checking*. *Formal Aspects of Computing* 24(2), pp. 249–266, doi:10.1007/s00165-011-0181-8.

[18] S. Minato (1993): *Fast Generation of Prime-Irredundant Covers from Binary Decision Diagrams*. *IEICE Transactions on Fundamentals of* E76-A(6), pp. 967–973.

[19] E. Morreale (1970): *Recursive Operators for Prime Implicant and Irredundant Normal Form Determination*. *IEEE Transactions on Computers* 19(6), pp. 504–509, doi:10.1109/T-C.1970.222967.

[20] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea & A. Tiwari (2004): *SAL 2*. In: *16th International Conference on Computer Aided Verification*, LNCS 3114, Springer, pp. 496–500, doi:10.1007/978-3-540-27813-9_45.

[21] F. Ortmeier, W. Reif & G. Schellhorn (2006): *Deductive Cause-Consequence Analysis (DCCA)*. *Proceedings of IFAC World Congress*.

[22] F. Ortmeier & G. Schellhorn (2007): *Formal Fault Tree Analysis - Practical Experiences*. *Electronic Notes in Theoretical Computer Science* 185, pp. 139 – 151, doi:10.1016/j.entcs.2007.05.034.

[23] Y. Papadopoulos & M. Maruhn (2001): *Model-Based Synthesis of Fault Trees from Matlab-Simulink Models*. In: *Proc. Int. Conf. on Dependable Systems and Networks (DSN 2001)*, IEEE Computer Society, pp. 77–82, doi:10.1109/DSN.2001.941393.

[24] D. Peled, T. Wilke & P. Wolper (1996): *An Algorithmic Approach for Checking Closure Properties of ω-Regular Languages*. In: *7th International Conference on Concurrency Theory*, LNCS 1119, Springer, pp. 596–610, doi:10.1016/S0304-3975(97)00219-3.

[25] A. Pnueli (1977): *The Temporal Logic of Programs*. In: *18th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, pp. 46–57, doi:10.1109/SFCS.1977.32.

[26] A. Rae & P. Lindsay (2004): *A Behaviour-based Method for Fault Tree Generation*. In: *Proceedings of 22nd Internatonal System Safety Conference*, System Safety Society, pp. 289–298.

Breaking Dense Structures – Proving Stability of Densely Structured Hybrid Systems*

Eike Möhlmann and Oliver Theel
Carl von Ossietzky University of Oldenburg
Department of Computer Science
D-26111 Oldenburg, Germany

{eike.moehlmann, theel}@informatik.uni-oldenburg.de

Abstraction and refinement is widely used in software development. Such techniques are valuable since they allow to handle even more complex systems. One key point is the ability to decompose a large system into subsystems, analyze those subsystems and deduce properties of the larger system. As cyber-physical systems tend to become more and more complex, such techniques become more appealing.
In 2009, Oehlerking and Theel presented a (de-)composition technique for hybrid systems. This technique is graph-based and constructs a Lyapunov function for hybrid systems having a complex discrete state space. The technique consists of (1) decomposing the underlying graph of the hybrid system into subgraphs, (2) computing multiple local Lyapunov functions for the subgraphs, and finally (3) composing the local Lyapunov functions into a piecewise Lyapunov function. A Lyapunov function can serve multiple purposes, e.g., it certifies stability or termination of a system or allows to construct invariant sets, which in turn may be used to certify safety and security.
In this paper, we propose an improvement to the decomposing technique, which relaxes the graph structure before applying the decomposition technique. Our relaxation significantly reduces the connectivity of the graph by exploiting super-dense switching. The relaxation makes the decomposition technique more efficient on one hand and on the other allows to decompose a wider range of graph structures.

Keywords: Hybrid Systems, Automatic Verification, Stability, Lyapunov Theory, Graphs, Relaxation

1 Introduction

In this paper, we present a relaxation technique for hybrid systems exhibiting dense graph structures. It improves the (de-)compositional technique proposed by Oehlerking and Theel in [10]. The relaxation results in hybrid systems that are well suited for (de-)composition. This increases the likeliness of successfully identifying Lyapunov functions.

Throughout the paper, in order to ease readability we will simply write "decomposition" or "decompositional technique" instead of "(de-)composition" or "(de-)compositional technique".

Stability, in general and for hybrid systems in particular, is a very desirable property, since stable systems are inherently fault-tolerant: after the occurrence of faults leading to, for example, a changed environment, the system will automatically "drive back" to the set of desired (i.e., stable) states. Stable systems are therefore particularly suited for contexts where autonomy is important such as for dependable assistance systems or in contexts where security has to be assured in an adverse environment.

Modeling such real world systems often involves the interaction of embedded systems (e.g., a controller) and its surrounding environment (e.g., a plant). Examples of such systems are automatic cruise

*This work has been partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

J. Pang, Y. Liu, and S. Mauw (Eds.): 4th International Workshop on Engineering Safety and Security Systems 2015 (ESSS'15)
EPTCS 184, 2015, pp. 49–63, doi:10.4204/EPTCS.184.4

controllers, engine control units, or unmanned powerhouses. In all these examples, an optimal operating range should be maintained. Although it is sometimes possible to discretize physical relations (using sampling) or to fluidize discrete steps (having a real-valued count of objects) it is more natural and less error-prone to use hybrid systems for modeling and verification. This is due to the fact that hybrid systems allow both: the representation of discrete and continuous behavior.

For hybrid systems with a complex discrete behavior, the technique proposed in [10] decomposes the monolithic problem of proving stability into multiple subproblems. But if a hybrid system exhibiting a complex control structure – in the sense of a dense graph structure – is decomposed, then the blow-up can be enormous. The result is a high number of subproblems that must be solved – this is not bad per se. But since the decompositional technique requires to underapproximate the feasible sets of each subproblem – when applied to often – results in the feasible set becoming empty. The relaxation technique presented in this paper reduces the number of steps required by the decomposition and, therefore, the number of underapproximations. This has two benefits: the runtime is reduced as well as the effect of underapproximations is minimized.

This paper is organized as follows. Section 2 gives a brief overview on related work. In Section 3, we define the hybrid system model, the stability property, an adaptation of the Lyapunov Theorem, and briefly sketch the idea of the decompositional proof technique. Section 4 describes our improvement to that proof scheme. In Section 5, we apply the relaxation to prove stability of three examples. The first example is the automatic cruise controller which is the motivating example for the decompositional technique. The second example is abstract and shows what happens if decomposition is applied to complete graph structures. The last example is a spidercam that exhibits a dense graph structure for which proving stability using decomposition is not possible. Finally, in Section 6, we give a short summary.

2 Related Work

In contrast to safety properties, stability has not yet received that much attention wrt. automatic proving and therefore, only a few tools are available. Indeed only the following automatic tools – each specialized for specific system classes – are known to the authors. Podelski and Wagner presented a tool in [12] which computes a sequence of snapshots and then tries to relate the snapshots in decreasing sequence. If successful, then this certifies region stability, i.e., stability with respect to a region instead of a single equilibrium point. Oehlerking et al. [9] implemented a powerful state space partitioning scheme to find Lyapunov functions for linear hybrid systems. The RSOLVER by Ratschan and She [18] computes Lyapunov-like functions for continuous system. Duggirala and Mitra [3] combined Lyapunov functions with searching for a well-foundedness relation for symmetric linear hybrid systems. Prabhakar and García [16] presented a technique for proving stability of hybrid systems with constant derivatives. Finally, some MATLAB toolboxes (YALMIP [5], SOSTOOLS [11]) that require a by-hand generation of constraint systems for the search of Lyapunov functions are available. These toolboxes do not automatically prove stability but assist in handling solvers.

Related theoretical works are the decompositional technique by Oehlerking and Theel [10], which we aim to improve, and the work on pre-orders for reasoning about stability in a series of papers by Prabhakar et al. [14, 13, 15] whose aim is a precise characterization of soundness of abstractions for stability properties. In contrast, our vision is an automatic computational engine for obtaining Lyapunov functions. The technique and tool presented in [16] is also based on abstractions. Unfortunately, their technique is restricted to hybrid systems whose differential equations have constant right hand sides

while our technique is more general. However, the techniques are not even mutually exclusive and have the potential to be combined.

3 Preliminaries

In this section, we give the definitions of the hybrid system model, global asymptotic stability, and discontinuous Lyapunov functions. Furthermore, we sketch the decomposition technique of [10].

Definition 1. *A **Hybrid Automaton** $\mathcal{H}$ is a tuple $(\mathcal{V},\mathcal{M},\mathcal{T},Flow,Inv)$ where*

- *$\mathcal{V}$ is a finite set of* variables *and $\mathcal{S}=\mathbb{R}^{|\mathcal{V}|}$ is the corresponding* continuous state space,
- *$\mathcal{M}$ is a finite set of* modes,
- *$\mathcal{T}$ is a finite set of* transitions *(m_1,G,U,m_2) where*
  - *$m_1,m_2\in\mathcal{M}$ are the* source and target mode *of the transition, respectively,*
  - *$G\subseteq\mathcal{S}$ is a* guard *which restricts the valuations of the variables for which this transition can be taken,*
  - *$U:\mathcal{S}\to\mathcal{S}$ is the* update function *which might update some valuations of the variables,*
- *$Flow:\mathcal{M}\to[\mathcal{S}\to\mathcal{P}(\mathcal{S})]$ is the* flow function *which assigns a* flow *to every mode. A flow $f\subseteq\mathcal{S}\to\mathcal{P}(\mathcal{S})$ in turn assigns a closed subset of $\mathcal{S}$ to each $\mathbf{x}\in\mathcal{S}$, which can be seen as the right hand side of a differential inclusion $\dot{\mathbf{x}}\in f(\mathbf{x})$,*
- *$Inv:\mathcal{M}\to\mathcal{P}(\mathcal{S})$ is the* invariant function *which assigns a closed subset of the continuous state space to each mode $m\in\mathcal{M}$, and therefore restricts valuations of the variables for which this mode can be active.*

A trajectory *of $\mathcal{H}$ is an infinite solution in form of a function $\tau(t)=(\mathbf{x}(t),m(t))$ over time t where $\mathbf{x}(\cdot)$ describes the evolution of the continuous variables and $m(\cdot)$ the corresponding evolution of the modes.*[1]

Roughly speaking, stability is a property basically expressing that all trajectories of the system eventually reach an equilibrium point of the sub-state space and stay in that point forever given the absence of errors. For technical reasons the equilibrium point is usually assumed to be the origin of the continuous state space, i. e. $\mathbf{0}$. This is not a restriction, since a system can always be shifted such that the equilibrium is $\mathbf{0}$ via a coordinate transformation. In the sequel, we focus on *asymptotic stability* which does not require the equilibrium point to be reached in finite time but only requires every trajectory to "continuously approach" it (in contrast to *exponential stability* where additionally the existence of an exponential rate of convergence is required).

In the following, we refer to $\mathbf{x}_{\downarrow\mathcal{V}'}\in\mathbb{R}^{|\mathcal{V}'|}$ as the sub-vector of a vector $\mathbf{x}\in\mathbb{R}^{\mathcal{V}}$ containing only values of variables in $\mathcal{V}'\subseteq\mathcal{V}$.

Definition 2 (Global Asymptotic Stability with Respect to a Subset of Variables [8])**.** *Let $\mathcal{H}=(\mathcal{V},\mathcal{M},\mathcal{T},Flow,Inv)$ be a hybrid automaton, and let $\mathcal{V}'\subseteq\mathcal{V}$ be the set of variables that are required to converge to the equilibrium point $\mathbf{0}$. A continuous-time dynamic system $\mathcal{H}$ is called* Lyapunov stable (LS) with respect to $\mathcal{V}'$ *if for all functions $\mathbf{x}_{\downarrow\mathcal{V}'}(\cdot)$,*

$$\forall\varepsilon>0:\ \exists\delta>0:\forall t\geq 0:||\mathbf{x}(0)||<\delta\Rightarrow\left|\left|\mathbf{x}_{\downarrow\mathcal{V}'}(t)\right|\right|<\varepsilon.$$

$\mathcal{H}$ is called globally attractive (GA) with respect to $\mathcal{V}'$ *if for all functions $\mathbf{x}_{\downarrow\mathcal{V}'}(\cdot)$,*

$$\lim_{t\to\infty}\mathbf{x}_{\downarrow\mathcal{V}'}(t)=\mathbf{0},\ i.e.,\forall\varepsilon>0:\exists t_0\geq 0:\forall t>t_0:\left|\left|\mathbf{x}_{\downarrow\mathcal{V}'}(t)\right|\right|<\varepsilon,$$

[1] Note, that definition of trajectories given here is for real time, i. e., $t\in\mathbb{R}_{\geq 0}$ while solutions of the relaxed hybrid automaton in Section 4 require a corresponding definition of trajectories for dense time, i. e., $t\in\mathbb{N}\times\mathbb{R}_{\geq 0}$. However, as there is only little difference in our setting and we do not directly reason about the solutions of the relaxation, we omit corresponding definitions.

where $\mathbf{0}$ *is the origin of* $\mathbb{R}^{|\mathcal{V}'|}$. *If a system is both globally stable with respect to* $\mathcal{V}'$ *and globally attractive with respect to* $\mathcal{V}'$, *then it is called* globally asymptotically stable (GAS) with respect to $\mathcal{V}'$.

Intuitively, LS is a boundedness condition, i. e., each trajectory starting δ-close to the origin will remain ε-close to the origin. GA ensures progress, i. e., for each ε-distance to the origin, there exists a point in time t_0 such that afterwards a trajectory always remains within this distance. It follows, that each trajectory is eventually always approaching the origin. This property can be proven using Lyapunov Theory [6]. Lyapunov Theory was originally restricted to continuous systems but has been lifted to hybrid systems.

Theorem 1 (Discontinuous Lyapunov Functions for a subset of variables [8])**.** *Let* $\mathcal{H} = (\mathcal{V}, \mathcal{M}, \mathcal{T}, Flow, Inv)$ *be a hybrid automaton and let* $\mathcal{V}' \subseteq \mathcal{V}$ *be the set of variables that are required to converge. If for each* $m \in \mathcal{M}$, *there exists a set of variables* $\mathcal{V}_m$ *with* $\mathcal{V}' \subseteq \mathcal{V}_m \subseteq \mathcal{V}$ *and a continuously differentiable function* $V_m : \mathcal{S} \to \mathbb{R}$ *such that*

1. *for each* $m \in \mathcal{M}$, *there exist two class* K^∞ *functions* α *and* β *such that*

$$\forall \mathbf{x} \in Inv(m) : \alpha\big(\big|\big|\mathbf{x}_{\downarrow \mathcal{V}_m}\big|\big|\big) \leq V_m(\mathbf{x}) \leq \beta\big(\big|\big|\mathbf{x}_{\downarrow \mathcal{V}_m}\big|\big|\big),$$

2. *for each* $m \in \mathcal{M}$, *there exists a class* K^∞ *function* γ *such that*

$$\forall \mathbf{x} \in Inv(m) : \dot{V}_m(\mathbf{x}) \leq -\gamma\big(\big|\big|\mathbf{x}_{\downarrow \mathcal{V}_m}\big|\big|\big)$$

   for each $\dot{V}_m(\mathbf{x}) \in \left\{ \left\langle \frac{dV_m(\mathbf{x})}{d\mathbf{x}} \;\middle|\; f(\mathbf{x}) \right\rangle \;\middle|\; f(\mathbf{x}) \in Flow(m) \right\}$,

3. *for each* $(m_1, G, U, m_2) \in \mathcal{T}$,

$$\forall \mathbf{x} \in G : V_{m_2}(U(\mathbf{x})) \leq V_{m_1}(\mathbf{x}),$$

then $\mathcal{H}$ *is globally asymptotically stable with respect to* $\mathcal{V}'$ *and* V_m *is called a* local Lyapunov function (LLF) *of m.*

In Theorem 1, $\left\langle \frac{dV(\mathbf{x})}{d\mathbf{x}} \;\middle|\; f(\mathbf{x}) \right\rangle$ denotes the inner product between the gradient of a Lyapunov function V and a flow function $f(\mathbf{x})$. Throughout the paper we denote by *mode constraints* the constraints of Type 1 and Type 2 and by *transition constraints* the constraints of Type 3.

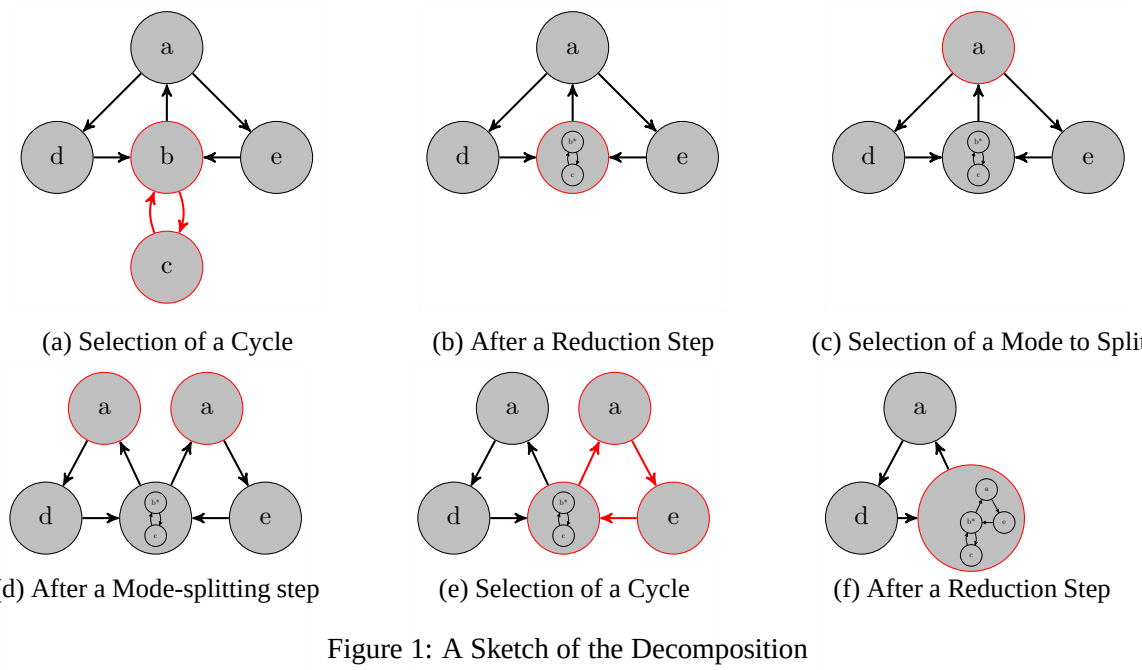
Decompositional Construction of Lyapunov Functions

In this section we briefly introduce the decompositional construction of Lyapunov functions for self-containment and refer to [10] for the details.

The decomposition technique introduces a so-called *constraint graph*. In the constraint graph, vertices are labeled with mode constraints and transition constraints for self-loops, i. e., $m_1 = m_2$ while edges are labeled with transition constraints for non-self-loops, i. e., $m_1 \neq m_2$. Obviously, any solution to the constraint graph is a solution to Theorem 1. The graph structure is exploited in two ways:

1) The constraint graph is partitioned into finitely many strongly connected components (SCCs). A trajectory entering an SCC of the corresponding hybrid automaton may either converge to 0 within the SCC or leave the SCC in finite time. In any case, once entered, an SCC might not be entered again. This allows us to compute LLFs for each SCC separately.

2) Each SCC is further partitioned into (overlapping) cycles. LLFs for modes in a cycle can also be computed separately but compatibility – wrt. constraints on the edges – has to be assured somehow. Compatibility can be guaranteed if the cycles are examined successively in the following way: A cycle



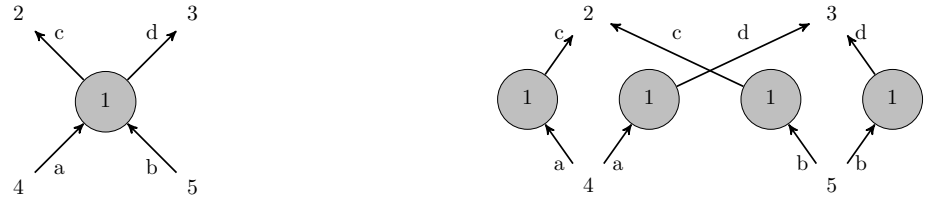
(a) Selection of a Cycle (b) After a Reduction Step (c) Selection of a Mode to Split

(d) After a Mode-splitting step (e) Selection of a Cycle (f) After a Reduction Step

Figure 1: A Sketch of the Decomposition

is selected and replaced by an underapproximation of the feasible set of its constraints, i. e., finitely many solutions (candidate LLFs) to the constraints of that cycle. Since the constraints describe a convex problem, conical combinations of the candidate LLFs satisfy the constraints, too. This step is called a *reduction step*. The reduction step collapses all vertices that lie only on that cycle and replaces references to LLFs in the constraints of adjacent edges by conical combinations of the candidate LLFs. This allows us to prove stability of each cycle separately while, cycle-by-cycle, ensuring compatibility of the feasible sets of the (overlapping) cycles.
The reduction step is visualized in Figure 1a and Figure 1b: In the former, a cycle is selected and in the latter, the cycle is replaced by a finite set of solutions of the corresponding optimization problem – visualized by collapsing the cycle into a single vertex.

The reduction step is more efficient if the cycle is connected to the rest of the graph by at most one vertex. We call such a cycle an *outer cycle* and the vertex a *border vertex*. On one hand, if the graph contains an outer cycle, then the cycle can be collapsed into a single vertex which replaces the border vertex. Thus, the feasible set of the cycle's constraints is replaced by a set of candidate LLFs. On the other hand, if the graph does not contain an outer cycle, then another step, called a *mode-splitting step*, is performed. In the mode-splitting step, a single vertex is replaced by a copy per pair of incoming and outgoing edges. This is visualized Figure 2. In Figure 2a, vertex 1 is connected to four other vertices by



(a) Before splitting (b) After splitting

Figure 2: The Mode-Splitting Step

two incoming and two outgoing edges. In Figure 2b, vertex 1 is replaced by four copies, where each one is connected to exactly one incoming and one outgoing edge. Depending on the order in which vertices are chosen for mode-splitting, one can make a cycle connected to the rest of the graph by exactly one vertex and then perform a reduction step. Clearly, the order of mode-splitting and reduction steps does not only affect the termination of the procedure, but also the size of the graph and, therefore, the number of cycles that have to be reduced. With a good order of reduction and mode-splitting steps, one ends up with a single cycle for which the following holds: The successful computation of candidate LLFs implies the existence of a piecewise Lyapunov function for the whole SCC.

Continuing on the example given in Figure 1: In Figure 1c, there are no outer cycles, thus, a mode-splitting step is performed: the vertex a is selected, copied twice, and each path is routed through one copy. The result is shown in Figure 1d. Since the result contains outer cycles, we can select an outer cycle as in Figure 1e and perform another reduction step resulting a single cycle being left. Figure 1f shows the result.

Automatically Computing Lyapunov Functions

To compute Lyapunov functions needed for decomposition as well as for the monolithic approaches each Lyapunov function is instantiated by a template involving free parameters. Using this Lyapunov function templates a constraint system corresponding to Theorem 1 is generated. Such a constraint system is then relaxed by a series of relaxations involving 1. the so-called *S-Procedure* [2] which restricts the constraints to certain regions and 2. the *sums-of-squares (SOS)* decomposition [17] which allows us to rewrite the polynomials as linear matrix inequalities (LMI). These LMIs in turn can be solved by Semidefinite Programming (SDP) [2]. Instances of solvers are CSDP [1] and SDPA [4]. These solvers typically use some kind of interior point methods and numerically approximate a solution. While this is very fast, such numerical solvers sometimes suffer from numerical inaccuracies. Therefore, constraints may be *strengthened* by adding additional "gaps". These gaps make the constraints more robust against numerical issues but sometimes result in the feasible set becoming empty.

These the gaps further limit the use of the decomposition as each reduction now "doubly" shrinks the feasible set: via gaps and via computing finitely many candidate LLFs.

4 Relaxation of the Graph Structure

In this section, we show how the decomposition can be improved by our graph structure-based relaxation. Consider the *underlying digraph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of a hybrid automaton with the set of vertices $\mathcal{V} = \mathcal{M}$ and the set of edges $\mathcal{E} = \{(m_1, m_2) \mid \exists (m_1, G, U, m_2) \in \mathcal{T}\}$. Note that the underlying graph has at most a single edge between any two vertices while the hybrid automaton might have multiple transitions between two modes. The *density* of the graph $\mathcal{G}$ is the fraction of the number of edges in the graph and the maximum possible number of edges in a graph of the same size, i. e., $\frac{|\mathcal{E}|}{|\mathcal{V}|(|\mathcal{V}|-1)}$.[2]

The idea is to identify a set of modes of a hybrid automaton whose graph structure is dense. This can, for example, be done by a clique-finding or dense-subgraph-finding algorithm. A *clique* is a complete subgraph, i. e., having a density of 1.[3] Our relaxation then rewires the transitions such that the resulting

[2] We are referring to the definition of density for directed graphs.

[3] Finding the maximum clique is NP-hard. However, a maximum clique is not required, any maximal clique (with more than two vertices) is sufficient. Even better as we are interested in dense structures only, we can use quasi cliques. A *quasi clique* is a subgraph where the density is not less than a certain threshold. Thus, any greedy algorithm can be used.

automaton immediately exhibits a structure well-suited for decomposition. By "well-suited," we mean that the graph structure contains mainly outer cycles.

The reason, that our relaxation technique plays so well with the decomposition technique, is as follows: if a hybrid system exhibits a dense graph structure, then the decomposition results in a huge blow-up. This blow-up is a result of the splitting step. The splitting step separates vertices shared between cycles, i. e., if there is more than one vertex shared between two or more cycles, then multiple copies are created. Thus, the higher the density of the graph structure is, the higher the blow-up gets. Further, if many cycles share many vertices – as in dense graphs – then whole cycles get copied and each copy requires solving an optimization problem and underapproximating the problem's feasible set. In contrast, our relaxation overapproximates the discrete behavior by putting each vertex in its own cycle and connecting this vertex by a new "fake" vertex. This reduces the number of optimization problems to be solved and the number of feasible sets to be underapproximated.

In the following, we define the relaxation operator. Then we give an algorithm which applies the relaxation integrated with decomposition. Finally, we prove termination and implication of stability of the hybrid automaton which has been relaxed.

Definition 3. *The graph structure relaxed hybrid automaton* $Rlx(\mathcal{H},\mathcal{M}_d) = (\mathcal{V}^\sharp,\mathcal{M}^\sharp,\mathcal{T}^\sharp,Flow^\sharp,Inv^\sharp) = \mathcal{H}^\sharp$ *of a hybrid automaton* $\mathcal{H} = (\mathcal{V},\mathcal{M},\mathcal{T},Flow,Inv)$ *wrt. the sub-component* $\mathcal{M}_d \subseteq \mathcal{M}$ *is defined as follows*

$$
\begin{aligned}
\mathcal{V}^\sharp &= \mathcal{V},\\
\mathcal{M}^\sharp &= \mathcal{M} \cup \{m_c\},\\
\mathcal{T}^\sharp &= \left\{ (m_1,G,U,m_2) \;\middle|\; \begin{matrix}(m_1,G,U,m_2)\in\mathcal{T},\\ \{m_1,m_2\}\cap\mathcal{M}_d = \emptyset\end{matrix} \right\}\\
&\quad \bigcup \left\{ \begin{matrix}(m_1,G,\mathrm{id},m_c),\\ (m_c,G,U,m_2)\end{matrix} \;\middle|\; \begin{matrix}(m_1,G,U,m_2)\in\mathcal{T},\\ \{m_1,m_2\}\cap\mathcal{M}_d \neq \emptyset\end{matrix} \right\},\\
Flow^\sharp(m) &= \begin{cases} \texttt{zero} & \text{if } m = m_c\\ Flow(m) & \text{otherwise,}\end{cases}\\
Inv^\sharp(m) &= \begin{cases} \emptyset & \text{if } m = m_c\\ Inv(m) & \text{otherwise,}\end{cases}
\end{aligned}
$$

where $\texttt{zero} : \mathcal{S} \to \mathcal{P}(\mathcal{S})$ *is a function assigning* $\mathbf{0}$ *to each* $\mathbf{x}\in\mathcal{S}$*, i. e.,* $\dot{\mathbf{x}} \in \{\mathbf{0}\}$.

In $\mathcal{T}^\sharp$ in Definition 3, we replace each transition $(m_1,G,U,m_2)\in\mathcal{T}$ connected to at least one mode in $\mathcal{M}_d$ with two transitions: one connecting the old source mode m_1 with the new mode m_c and the other connecting m_c with the old target mode m_2. We call this step a *transition-splitting step* where the result is a pair of transitions which is called *split transition* and the set of all split transitions is denoted by ST.

Intuitively, the introduced mode m_c is a dummy mode whose invariant always evaluates to false and the flow function does not change the valuations of the continuous variables. Indeed, the mode cannot be entered and thus, a trajectory taking an ingoing transition must, immediately, take an outgoing transition. The sole reason to add the mode is changing the structure of the hybrid system's underlying graph: the new structure contains mainly cycles that are connected via m_c.

Next, we show how to integrate decomposition and relaxation. Pseudo-code of the relaxation function and a reconstruction function – which step-by-step reverts the relaxation – can be found in Algorithm 1 and Algorithm 2, respectively. Algorithm 3 gives pseudo-code of the main algorithm. The main al-

Algorithm 1: The Relaxation Function

input : A hybrid automaton $\mathcal{H}$, a dense sub-component $\mathcal{M}_d$ of $\mathcal{H}$.
output: The relaxed version of $\mathcal{H}$, a set of split transitions ST, the central mode m_c.

1 $m_c \leftarrow$ `newMode();`
2 $\mathcal{H}.\mathcal{M} \leftarrow \mathcal{H}.\mathcal{M} \cup \{m_c\}$;
3 $\mathcal{H}.Flow(m_c) \leftarrow$ `zero`;
4 $\mathcal{H}.Inv(m_c) \leftarrow \emptyset$;
5 $\mathcal{T} \leftarrow \mathcal{H}.\mathcal{T}$;
6 **foreach** $t = (m_1, G, U, m_2) \in \mathcal{T}$ **do**
7 **if** $\{m_1, m_2\} \cap \mathcal{M}_d \neq \emptyset$ **then**
  `// split the transitions into two parts`
8 $t_1 \leftarrow (m_1, G, \mathrm{id}, m_c)$;
9 $t_2 \leftarrow (m_c, G, U, m_2)$;
  `// replace the transition by the two parts`
10 $\mathcal{H}.\mathcal{T} \leftarrow (\mathcal{H}.\mathcal{T} \setminus \{t\}) \cup \{t_1, t_2\}$;
  `// keep account of split transitions`
11 $ST \leftarrow ST \cup \{(t_1, t_2)\}$

gorithm works as follows: Step 1) The function `relax` relaxes the graph structure of the hybrid automaton $\mathcal{H}$ and generates the set of split transitions ST. Step 2) If the set ST is empty, then call `applyDecomposition` with the original automaton and return the result – this function applies the original decompositition technique as described in Section 3. Step 3) Otherwise, apply `applyDecomposition` on the current relaxed form of the automaton. If the result is `stable`, then return the result. Otherwise, if the original decompositional technique has failed, then it returns a *failed subgraph* that is a subgraph for which it was unable to find Lyapunov functions. Step 4) Choose a split transition from the set ST which also belongs to the failed subgraph. It is then used to reconstruct a transition from the original hybrid automaton. Then execution is continued with step 2. Step 5) If no such split transition exists, then the algorithm fails and returns the failed subgraph since this failing subgraph will persist in the automaton. Further reverting the relaxation cannot help because no split transition is contained in the failed subgraph.

Next, we prove termination and soundness of the algorithm. Here, soundness indicates that a Lyapunov function-based stability certificate for a relaxed automaton implies stability of the original, unmodified automaton. In particular, the local Lyapunov functions of the relaxed hybrid automaton are valid local Lyapunov functions for the original automaton.

Termination of the Integrated Algorithm

Theorem 2. *The proposed algorithm presented in Algorithm 3 terminates.*

Proof. The function `relax` terminates since the copy of the set of transitions of $\mathcal{H}$ is finite and is not modified in the course of the algorithm. The while-loop terminates if either an `applyDecomposition` is successful, no pair for reconstruction can be identified, or the set ST is empty. In the first two cases, the algorithm terminates directly. For the last case, we assume that no call to `applyDecomposition` is successful and a spilt transition is always found. Then, in each iteration of the loop, one edge is removed

Algorithm 2: The Reconstruction Function

input : A relaxed hybrid automaton $\mathcal{H}$, a set of split transitions ST, a pair of split transitions (t_1,t_2), where $t_1 = (m_1, G, \mathrm{id}, m_c)$, $t_2 = (m_c, G, U, m_2)$ and m_c is the central mode.

output: A relaxed hybrid automaton $\mathcal{H}$ with one split transition being reconstructed, the set of split transitions ST

`// reconstruct the original transition`

1 $t \leftarrow (m_1, G, U, m_2)$;

`// replace the split transition` (t_1,t_2) `by` t

2 $\mathcal{H}.\mathcal{T} \leftarrow (\mathcal{H}.\mathcal{T} \setminus \{t_1,t_2\}) \cup \{t\}$;

`// update the set of split transitions`

3 $ST \leftarrow ST \setminus \{(t_1,t_2)\}$;

`// remove` m_c `iff unconnected`

4 **if** $ST = \emptyset$ **then**

5 $\quad \mathcal{H}.\mathcal{M} \leftarrow \mathcal{H}.\mathcal{M} \setminus \{m_c\}$;

from ST. The set ST is finite because the relaxation function `relax` splits only finitely many edges. Thus, the set ST becomes eventually empty. Therefore, the loop terminates. □

Preservation of Stability

Theorem 3. *For any hybrid automaton $\mathcal{H}$ and a sub-component $\mathcal{M}_d$, it holds: If a family of local Lyapunov functions (V_m) proving $Rlx(\mathcal{H},\mathcal{M}_d)$ to be GAS exists, then there exists a family of local Lyapunov functions for $\mathcal{H}$ proving $\mathcal{H}$ to be GAS.*

Proof. Given a hybrid automaton $\mathcal{H} = (\mathcal{V},\mathcal{M},\mathcal{T},Flow,Inv)$. Let $Rlx(\mathcal{H},\mathcal{M}_d) = (\mathcal{V}^\sharp,\mathcal{M}^\sharp,\mathcal{T}^\sharp,Flow^\sharp,Inv^\sharp) = \mathcal{H}^\sharp$, be a graph structure-relaxed version of $\mathcal{H}$ where $\mathcal{M}_d \subseteq \mathcal{M}$ is the sub-component of $\mathcal{H}$ that has been relaxed. Further, let (V_m) be the family of local Lyapunov functions that prove stability of $\mathcal{H}^\sharp$ and let ST be the set of split transitions – some transition may have been reconstructed. Now, it must be shown that V_m are valid Lyapunov functions for $\mathcal{H}$.
The mode constraints of Theorem 1 trivially hold, since *Rlx* alters neither the flow functions nor the invariants, i. e., $\forall m \in \mathcal{M} : Flow^\sharp(m) = Flow(m) \wedge Inv^\sharp(m) = Inv(m)$. The transition constraint also holds for all transitions that are not altered by *Rlx* or have been reconstructed, i. e., $\mathcal{T} \cap \mathcal{T}^\sharp$. Now assume that $t \in \mathcal{T} \setminus \mathcal{T}^\sharp$ is an arbitrary transition for which the transition constraint does not hold. We show that this leads to a contradiction. Due to the definition of *Rlx* all transition in $\mathcal{T} \setminus \mathcal{T}^\sharp$ are split transitions and there is a corresponding pair in ST. Let $(t_1,t_2) \in ST$ be the pair corresponding to $t = (m_1,G,U,m_2)$. Since (V_m) is a valid family of local Lyapunov function for $\mathcal{H}^\sharp$, the transition constraint holds for all transitions in $\mathcal{T}^\sharp$. In particular, the transition constraint holds for $t_1 = (m_1,G,\mathrm{id},m_c)$ and $t_2 = (m_c,G,U,m_2)$. Thus,

$$\forall x \in G : V_{m_c}(\mathrm{id}(x)) \leq V_{m_1}(x) \wedge \forall x \in G : V_{m_2}(U(x)) \leq V_{m_c}(x).$$

It follows, that

$$\forall x \in G : V_{m_2}(U(x)) \leq V_{m_c}(x) \leq V_{m_1}(x).$$

Therefore, the transition constraint holds for t. But this contradicts the assumption. ↯ □

While Theorem 3 shows that stability of the relaxed automaton yields stability of the original automaton, the contrary is not true. Figure 3 shows a hybrid system where the relaxation renders the system unstable.

Algorithm 3: The Integrated Relaxation and Decomposition Algorithm

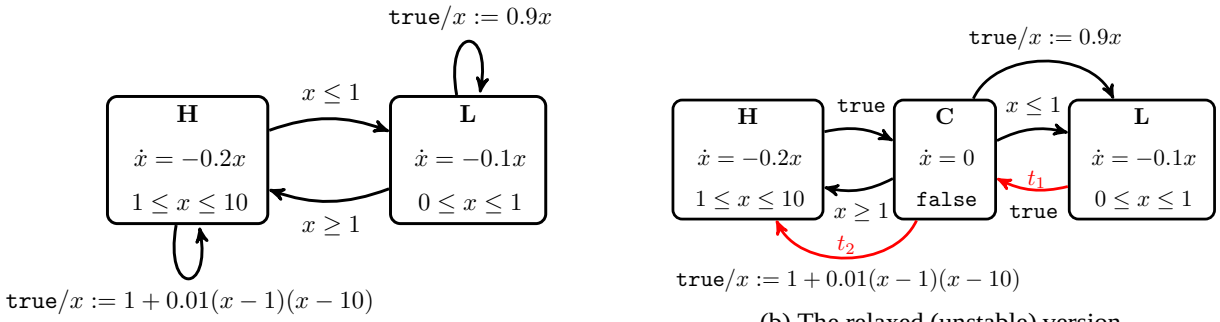
input : A hybrid automaton $\mathcal{H}$, a set of modes $\mathcal{M}_d$ corresponding to a dense subgraph.
output: **stable** if the $\mathcal{H}$ is stable and **failed** otherwise.

```
// relax the graph structure
1  H, ST, m_c ← relax(H, M_d);
2  while ST ≠ ∅ do
       // apply decomposition
3      result ← applyDecomposition(H);
4      if result is stable then
5          return stable;
       // apply reconstruction
6      if ∃(t1, t2) ∈ ST : {t1, t2} ∩ failedSubgraph(result) ≠ ∅ then
7          H, ST ← reconstruct(H, ST, (t1, t2), m_c);
8      else
9          return result;
   // apply decomposition on the original automaton
10 result ← applyDecomposition(H);
11 return result;
```

This example exploits that the relaxation may introduce spurious trajectories. This happens if there are transitions with overlapping guard sets connected to the central mode m_c. A trajectory of the relaxed automaton might then take the first part of a split transition to the central mode m_c and continues with the second part of a different split transition. A transition corresponding to this behavior might not exist in the unmodified hybrid automaton. While this does not render our approach being incorrect, it may lead to difficulties since these extra trajectories have to be GAS, too. In case of the system in Figure 3, new trajectories are introduced which allow a trajectory to jump back from the mode `L` to `H` by taking the transitions t_1, t_2. This behavior corresponds to leaving `L` by the right self-loop and entering `H` by the left self-loop, which is obviously impossible. However, due to the update the value of x might increase as $1 + 0.01(x-1)(x-10) > 1$ for $x < 1$.

In general our relaxation introduces conservatism which is again reduced step-by-step by the reconstruction. The degree of conservatism highly depends on the guards of the transitions since the central mode relates all LLFs of modes in $\mathcal{M}_d$. Therefore, if more guards are overlapping, more LLFs have to be compatible even if not needed in the original automaton.

One possibility to counter-act this issue is to introduce a new continuous variable in the relaxed automaton which is set to a unique value per split transition: the update function of the first part of a split transition sets the value used to guard the second part of the transition. Indeed, this trick discards any spurious trajectories for the price of an additional continuous variable. However, since the values of that variable are somewhat artificial, a Lyapunov function may not make use of that variable. Thus, this trick will not ease satisfying the conditions of the Lyapunov theorem in general.



(a) The unmodified (stable) version.

(b) The relaxed (unstable) version.

Figure 3: A Hybrid System; Unstable after Relaxation

Graph Structure	Nodes (n)	Edges	Decomposition			With Relaxation	
			Reductions	Mode-Splittings	Time	Reductions	Time
directed K_1	1	0	0	0	0.04s	0	0.04s
directed K_2	2	2	1	0	0.04s	2	0.04s
directed K_3	3	6	6	4	0.21s	3	0.05s
directed K_4	4	12	47	25	1.15s	4	0.05s
directed K_5	5	20	1852	352	13h22m	5	0.05s
Spidercam	9	32	753	287	1h46m	9	0.06s
Cruise Controller	6	11	7	6	0.060s	6	0.06s

Table 1: Comparison of the Decomposition with and without Relaxation

5 Application of the Relaxation

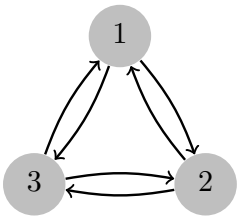
In this section we present three examples where the graph structure-based relaxation suggested in this paper improves the application of the decomposition technique. The first example deals with the automatic cruise controller (ACC) of [8]. The second example is the fully connected digraph K_3. The K_3 does not represent a concrete hybrid automaton but a potential graph structure of a hybrid automaton. The last example is a spidercam. Here, the graph is not as fully connected as the K_3 example, but its density is already too high to apply decomposition directly.

We have implemented the decomposition and relaxation in python. Table 1 gives the graph properties and a comparison of the number of reduction steps required by the decomposition with and without relaxation (in the best case). The given data was obtained without actually computing Lyapunov functions focusing on the graph related part of the decomposition. In fact, computing Lyapunov functions for the spidercam via decomposition without our relaxation fails after 18 steps.

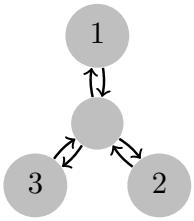
Example 1: The Automatic Cruise Controller (ACC)

The automatic cruise controller (ACC) regulates the velocity of a vehicle. Figure 5 shows the controller as an automaton. The task of the controller is to approach a user-chosen velocity – indeed the variable v represents the velocity relative to the desired velocity.

The ACC is globally asymptotically stable. It can be proven stable using the original decomposition technique (cf. [8, 7]). Indeed, the graph structure is sparse and thus, already well-suited for applying the decomposition technique directly. In fact, only one more cycle needs to be reduced compared to decomposition after relaxation (cf. Table 1). Even though the relaxation is not needed here, it also does not harm, though, it may be used for sparse graphs structures, too.



(a) original



(b) relaxed

Figure 4: The K_3.

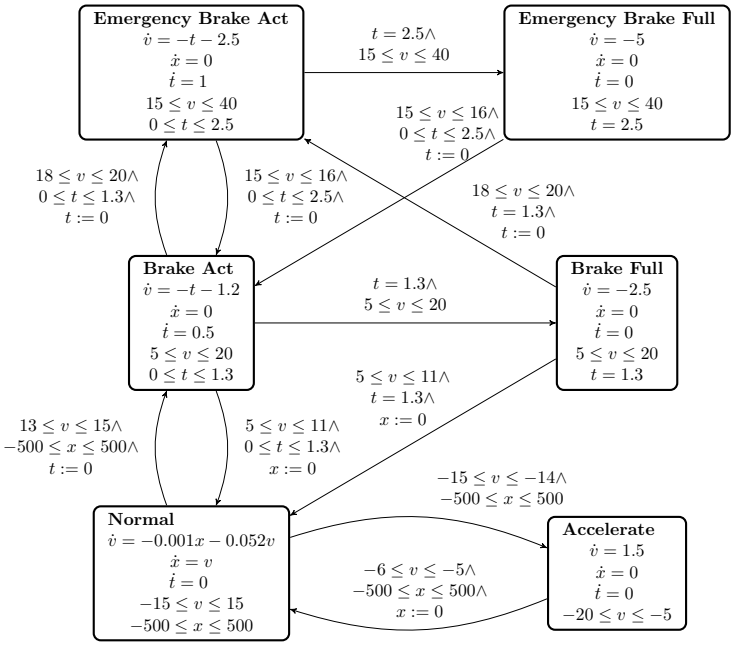


Figure 5: The Automatic Cruise Controller [8]

Example 2: The directed K_3

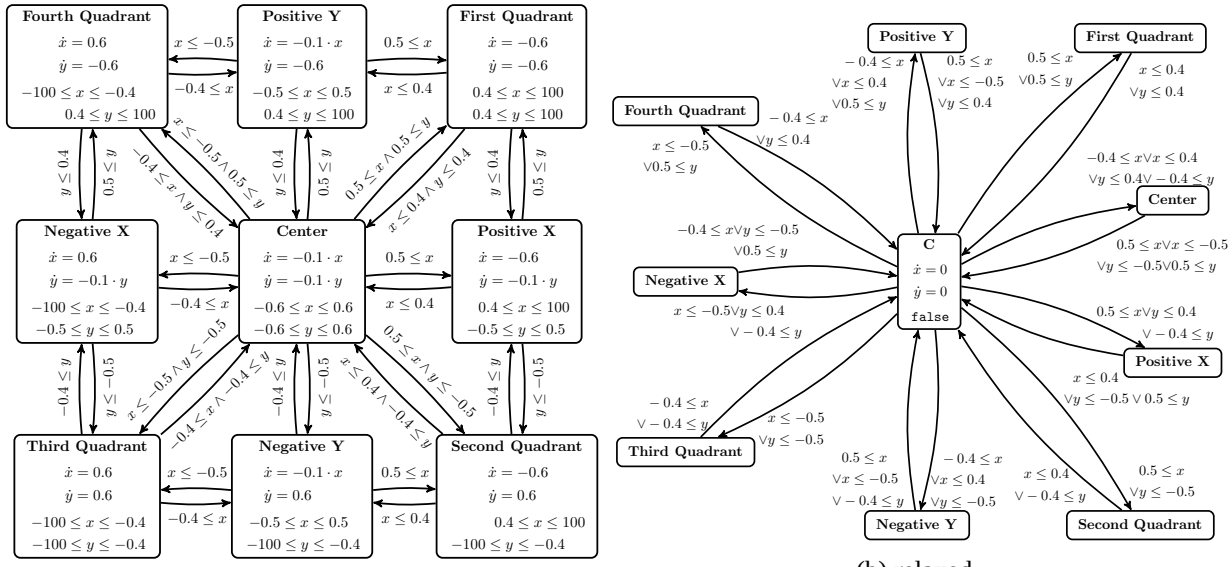
The directed K_n is a fully connected digraph with n nodes. The K_3 as well as a relaxed version of it is shown in Figure 4. In a fully connected digraph, there is a single edge from each node to each other node, resulting in a total number of $n(n-1)$ edges. The number of cycles, the decomposition technique has to reduce, grows very fast with n which can be seen in Table 1. In comparison, the number of cycles in the relaxed version of the graph grows linearly with n, assuming that the edges can be concentrated[4]. Otherwise, after the relaxation, each original node has $n-1$ incoming and $n-1$ outgoing edges where each edge connects the node with the central node m_c. Each such combination forms a cycle between m_c and an original mode, giving a total of $n(n-1)(n-1)$ cycles in the worst case. This cubic growth is still much less than the number of reductions without relaxation.

Such a graph might not be the result of a by-hand designed system but might be the outcome of a synthesis or an automatic translation. However, the fast growth of the cycles also indicates the high number of reduction and therefore underapproximations.

Example 3: The Spidercam

A spidercam is a movable robot equipped with a camera. It is used at sport events such as a football matches. The robot is connected to four cables. Each cable is attached to a motor that is placed high above the playing field in a corner of a stadium. By winding and unwinding the cables – and thereby controlling the length of the cables, – the spidercam is able to reach nearly any position in the three-dimensional space above the playing field. Figure 6 shows a very simple model of such a spidercam in the plane. The target is to stabilize the camera at a certain position. The continuous variables x and y denote the distance relative to the desired position on the axis induced by the cables.

[4]With "concentrating edges," we mean that edges with the same source and target node are handled as a single edge for the cycle finding algorithm.



(a) unmodified

(b) relaxed

Figure 6: The Simple Planar Spidercam

In the model, we assume a high-level control of the motor engines, i. e., the movement is on axis $\dot{x}$ and $\dot{y}$ instead of a low-level control of each individual motor. The model has nine modes: one mode that controls the behavior while being close to the desired position, four modes corresponding to nearly straight movements along one of the axes and four modes cover the quadrants between the axes. The maximal velocity in the direction of each axis is limited from above by $0.6\frac{m}{s}$. Thus, in the four modes corresponding to the quadrants, the movement in each direction is at full speed. In the four modes corresponding to the axes, the movement on the particular axis is at full speed while the movement orthogonal to the axis is proportional to the distance. In the last mode, the speed in both directions is proportional to the distance.

The spidercam is globally asymptotically stable which can be proven fully automatically. However, it is not possible to obtain a piecewise Lyapunov function via decomposition without relaxation due to accumulating underapproximations of the partial solutions and the high number of cycles that have to be reduced.[5] The reason is that each time a cycle is reduced, the feasible set of a subproblem is underapproximated by a finite set of solutions which finally results in a feasible set becoming empty and no LLFs can be found.

In contrast, relaxing the graph structure followed by applying the decomposition is successful immediately. In particular, no reconstruction step is required.

6 Summary

We have presented a relaxation technique based on the graph structure of a hybrid automaton. The relaxation exploits super-dense switching or cascaded transitions to modify the transitions of the hybrid automaton in a way that improves the decompositional proof technique of [10]. The idea is to re-route

[5] We used the implementation in STABHYLI [7] which currently does not contain strategies to handle the situation where no reduction is possible. The current implementation would then simply fail. Even though, it is theoretically possible to perform some form of backtracking, it is hard to decide which underapproximation must be refined.

every transition through a new “fake” node. Thus, if in the original automaton a single transition is taken, then the relaxed automaton has to take the cascade of two transitions to achieve the same result. However, the relaxed automaton’s graph structure is better suited towards decomposition. Furthermore, the procedure can be automated which is very much desired as our focus is the automation of Lyapunov function-based stability proofs. Furthermore, in Section 5, we successfully employed the proposed technique in some examples.

The decompositional proof technique is particularly well-suited to prove stability of large-scale hybrid systems because it allows: 1. to decompose a monolithic proof into several smaller subproofs, 2. to reuse subproofs after modifying the hybrid system, and 3. to identify critical parts of the hybrid automaton. All these benefits are not available when the hybrid system exhibits a very dense graph structure of the automaton because that would lead to an enormous number of computational steps required in the decomposition. The proposed relaxation overcomes these matters in the best case. If the relaxation is too loose, then our technique falls back to step-by-step reconstruct the original automaton. Each step increases the effort needed for the decomposition until a proof succeeds or ultimately – in the worst case – the original automaton gets decomposed. Future research will include a tighter coupling of the decomposition and our relaxation approach. A first step will be to not discard the progress made by the decomposition but reuse the “gained knowledge”. Doing so will greatly reduce the computational effort.

References

[1] Brian Borchers (1999): *CSDP, a C Library for Semidefinite Programming. Optim. Met. Softw.* 10, pp. 613–623, doi:10.1080/10556789908805765.

[2] Stephen Boyd & Lieven Vandenberghe (2004): *Convex Optimization*. Cambridge University Press, doi:10.1017/CBO9780511804441.

[3] Parasara Sridhar Duggirala & Sayan Mitra (2012): *Lyapunov abstractions for inevitability of hybrid systems*. In: *Proceedings of the 15th International Conference on Hybrid Systems: Computation and Control (HSCC’12)*, pp. 115–124, doi:10.1145/2185632.2185652.

[4] Katsuki Fujisawa, Kazuhide Nakata, Makoto Yamashita & Mituhiro Fukuda (2007): *SDPA Project : Solving Large-Scale Semidefinite Programs*. *JORSP* 50(4), pp. 278–298. Available at `http://ci.nii.ac.jp/naid/110006532053/en/`.

[5] J. Löfberg (2004): *YALMIP : A Toolbox for Modeling and Optimization in MATLAB*. In: *Proceedings of the 13th Conference on Computer-Aided Control System Design (CACSD’04)*, Taipei, Taiwan, doi:10.1109/CACSD.2004.1393890.

[6] M.A. Lyapunov (1907): *Problème général de la stabilité du movement*. In: *Ann. Fac. Sci. Toulouse, 9*, Université Paul Sabatier, pp. 203–474, doi:10.5802/afst.246. (Translation of a paper published in Comm. Soc. Math. Kharkow, 1893, reprinted Ann. Math. Studies No. 17, Princeton Univ. Press, 1949).

[7] Eike Möhlmann & Oliver E. Theel (2013): *Stabhyli: A Tool for Automatic Stability Verification of Non-Linear Hybrid Systems*. In: *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control (HSCC’13)*, pp. 107–112, doi:10.1145/2461328.2461347.

[8] Jens Oehlerking (2011): *Decomposition of Stability Proofs for Hybrid Systems*. Ph.D. thesis, Carl von Ossietzky University of Oldenburg, Department of Computer Science, Oldenburg, Germany.

[9] Jens Oehlerking, Henning Burchardt & Oliver E. Theel (2007): *Fully Automated Stability Verification for Piecewise Affine Systems*. In: *Proceedings of the 10th international conference on Hybrid systems: computation and control (HSCC’07)*, pp. 741–745, doi:10.1007/978-3-540-71493-4_74.

[10] Jens Oehlerking & Oliver E. Theel (2009): *Decompositional Construction of Lyapunov Functions for Hybrid Systems*. In: *Proceedings of the 12th International Conference on Hybrid Systems: Computation and Control (HSCC'09)*, pp. 276–290, doi:10.1007/978-3-642-00602-9_20.

[11] A. Papachristodoulou, J. Anderson, G. Valmorbida, S. Prajna, P. Seiler & P. A. Parrilo (2013): *SOSTOOLS: Sum of squares optimization toolbox for MATLAB*. Available at `http://arxiv.org/abs/1310.4716`.

[12] Andreas Podelski & Silke Wagner (2007): *Region Stability Proofs for Hybrid Systems*. In: *Formal Modelling and Analysis of Timed Systems (FORMATS'07)*, pp. 320–335, doi:10.1007/978-3-540-75454-1_23.

[13] Pavithra Prabhakar (2012): *Foundations for approximation based analysis of stability properties of hybrid systems*. In: *Proceedings of the 50th Annual Allerton Conference on Communication, Control, and Computing*, pp. 1602–1609, doi:10.1109/Allerton.2012.6483412.

[14] Pavithra Prabhakar, Geir E. Dullerud & Mahesh Viswanathan (2012): *Pre-orders for reasoning about stability*. In: *Proceedings of the 15th International Conference on Hybrid Systems: Computation and Control (HSCC'12)*, pp. 197–206, doi:10.1145/2185632.2185662.

[15] Pavithra Prabhakar, Jun Liu & Richard M. Murray (2013): *Pre-orders for reasoning about stability properties with respect to input of hybrid systems*. In: *Proceedings of the International Conference on Embedded Software (EMSOFT'13)*, pp. 1–10, doi:10.1109/EMSOFT.2013.6658602.

[16] Pavithra Prabhakar & Miriam Garcia Soto (2013): *Abstraction Based Model-Checking of Stability of Hybrid Systems*. In: *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*, pp. 280–295, doi:10.1007/978-3-642-39799-8_20.

[17] S. Prajna & A. Papachristodoulou (2003): *Analysis of Switched and Hybrid Systems - Beyond Piecewise Quadratic Methods*. In: *American Control Conference, 2003. Proceedings of the 2003*, 4, pp. 2779–2784 vol.4, doi:10.1109/ACC.2003.1243743.

[18] Stefan Ratschan & Zhikun She (2010): *Providing a Basin of Attraction to a Target Region of Polynomial Systems by Computation of Lyapunov-Like Functions*. *SIAM J. Control and Optimization* 48(7), pp. 4377–4394, doi:10.1137/090749955.

Formal Verification of Real-Time Function Blocks Using PVS

Linna Pang, Chen-Wei Wang, Mark Lawford, Alan Wassyng
McMaster Centre for Software Certification, McMaster University, Canada L8S 4K1
{pangl,wangcw,lawford,wassyng}@mcmaster.ca

Josh Newell, Vera Chow, and David Tremaine
Systemware Innovation Corporation, Toronto, Canada M4P 1E4
{jnewell,vchow,tremaine}@swi.com

A critical step towards certifying safety-critical systems is to check their conformance to hard real-time requirements. A promising way to achieve this is by building the systems from pre-verified components and verifying their correctness in a compositional manner. We previously reported a formal approach to verifying function blocks (FBs) using tabular expressions and the PVS proof assistant. By applying our approach to the IEC 61131-3 standard of Programmable Logic Controllers (PLCs), we constructed a repository of precise specification and reusable (proven) theorems of feasibility and correctness for FBs. However, we previously did not apply our approach to verify FBs against timing requirements, since IEC 61131-3 does not define composite FBs built from timers. In this paper, based on our experience in the nuclear domain, we conduct two realistic case studies, consisting of the software requirements and the proposed FB implementations for two subsystems of an industrial control system. The implementations are built from IEC 61131-3 FBs, including the on-delay timer. We find issues during the verification process and suggest solutions.

1 Introduction

Many industrial safety-critical software control systems are based upon Programmable Logic Controllers (PLCs). Function blocks (FBs) are reusable components for implementing the behaviour of PLCs in a hierarchical way. In one of its supplements, the aviation standard DO-178C [1] advocates the use of formal methods to construct, develop, and reason about mathematical models of system behaviours. Applying the principles of DO-178C to PLC-based systems, we may obtain high-quality PLCs by: 1) pre-verifying standard FBs using formal methods; 2) building systems from pre-verified components; and 3) verifying their correctness in a compositional manner.

We recently reported a formal methodology [10] for specifying requirements for FBs, and for verifying the correctness of their implementations expressed in, e.g., function block diagrams (FBDs). In our approach, we use tabular expressions (a.k.a. function tables) [12] for specification and the PVS proof assistant [9] for formal verification. Tabular expressions are a way to document system requirements as black-box, input-output relations that has proven to be practical and effective in industry [14]. PVS provides an integrated environment with mechanized support for writing specifications using tabular expressions and (higher-order) predicates, and for (interactively) proving that implementations satisfy the tabular requirements using sequent-style deductions. We successfully applied our approach to the FB library of IEC 61131-3 [6, Annex F], an industrial standard for PLCs, resulting in a repository of: 1) precise specifications of input-output requirements; and 2) reusable theorems of feasibility and correctness for the FB library.

A critical step towards certifying safety-critical systems is to check their conformance to hard real-time requirements. An *implementable* timing requirement must specify *tolerances* to account for various

J. Pang, Y. Liu, and S. Mauw (Eds.): 4th International Workshop on Engineering Safety and Security Systems 2015 (ESSS'15)
EPTCS 184, 2015, pp. 65–79, doi:10.4204/EPTCS.184.5

factors — e.g., sampling rates, computation time, and latency — that will delay the software controller's response to its operating environment. A common type of functional timing requirements specifies that a monitored condition C must sustain over a time duration, say *timeout*, with tolerances $-\delta L$ and $+\delta R$, before being detected by the controller. Such sustained timing requirements may be formalized using an infix *Held_For* operator [15]. For example, we write

$$(\ (signal \geq setpoint)\ Held\_For\ (300,\ -50,\ +50)\ ) \Rightarrow (c\_var = trip) \tag{1}$$

to specify that a sensor signal going out of its safety range should cause a "trip" if it sustains for longer than 350 ms, and should not if it lasts for less than 250 ms (to filter out the effect of a noisy signal).

The requirement specified in Eq. 1 is *non-deterministic* since it allows any implementation that trips when $signal \geq setpoint$ sustains between $[250ms,\ 350ms]$. As we will see in our two case studies, such a simple requirement can be used as part of specifying more complex real-time behaviour. To resolve such non-determinism, at the requirements level we adopt a deterministic operator *Held_For_I* [4, p. 86], which becomes *true* at the first sampling point after the monitored condition has been enabled for $d - \delta L$ time units. For example, by substituting the expression $((signal \geq setpoint)\ Held\_For\_I\ (300 - 50))$ into Eq. 1, we specify that the triggering condition sustaining for longer than 250 ms should cause a "trip". Similarly, at the implementation level we adopt the *Timer_I* operator [4, p. 98] for counting the elapsed time of some monitored condition. The relationship between these two operators, at levels of requirements and implementation, is proved as a theorem *TimerGeneral_I* [4, p. 99]: $(C\ Held\_For\_I\ (timeout - \delta L))$ is equivalent to $(Timer\_I\ (C) \geq timeout - \delta L)$. See also Sec. 2.

We previously did not apply our approach [10] to verify FBs against this type of more complex timing requirements because IEC 61131-3 only includes simple timer blocks (i.e, on-delay, off-delay, and pulse timers), but not any more complex FBs built from those timers. Furthermore, our requirements model for IEC 61131-3 timers [10] describes idealized behaviour: as the monitored condition becomes enabled, the timer instantaneously responds (i.e., starts counting the duration of enablement), not considering sampling, computational delays, and timing tolerances.

Based on our experience on the Darlington Nuclear Shutdown Systems Trip Computer Software Redesign Project [14], and motivated by anticipated FB based projects, we address the above issues by conducting realistic case studies. Each case study consists of the software requirements and FBD implementation for a subsystem of an industrial control system. Implementations are built from IEC 61131-3 FBs, including the on-delay timer to implement more complex real-time behaviour.

Fig. 1 summarizes our verification process and contributions. To incorporate the notion of tolerances, we reuse the timing operators *Held_For_I* (to formalize requirements) and *Timer_I* (to formalize implementations) from [4]. The verification goal is that the proposed FBD implementations, included in the Software Design Description (SDD), are: (a) *consistent*, or feasible, meaning that an output can always be produced on valid inputs; and (b) *correct* with respect to the timing requirements specified using *Held_For_I*, included in the Software Requirements Specification (SRS). This work builds on our previous results of verifying IEC 61131-3 FBs [10] that provide a sound semantic foundation for formalizing and verifying PLC programs expressed using FBDs.

There are four contributions of this paper. First, to incorporate tolerances, we use the *Timer_I* operator to re-formalize all three IEC 61131-3 timer (Sec. 3). Second, for the representative subsystems we study (one with a feedback loop presented in Sec. 4 and the other in Sec. 5), we use the re-formalized IEC 61131 timers for their proposed FBD implementations, and prove that they are feasible and satisfy the intended timing requirements in SRS. Third, we find issues of initialization failure (Sec. 4) and missing implementation assumptions (Sec. 5), and suggest possible solutions. Fourth, we identify pat-

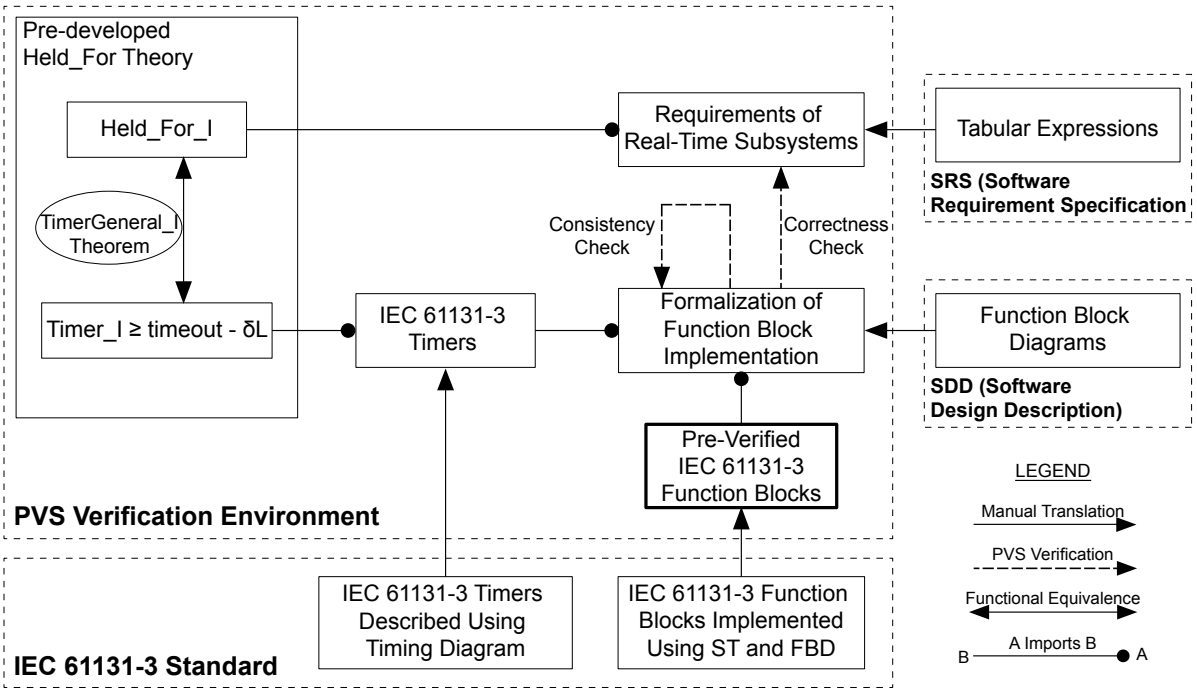


Figure 1: Framework for verification of FB based systems timing requirements

terns of proof commands (Sec. 6) that are amenable to strategies (or proof scripts) that will facilitate the automated verification of the feasibility and correctness of other subsystems.

Resources. Sources of the case studies (verified using PVS 6.0) are available at `http://www.cas.mcmaster.ca/~lawford/papers/ESSS2015`. Background theories (e.g., *Held_For_I*, *Timer_I*, etc.) and complete details of case studies covered in this paper are included in an extended report [11].

2 Preliminaries

We review the use of tabular expressions, the relevant PVS theories of timing operators at levels of requirements and implementation, and the formal verification approach [10] that is adapted in our two case studies (Sec. 4 and Sec. 5).

2.1 Tabular Expressions Tabular expressions (a.k.a. function tables) [12] are an effective approach for describing conditionals and relations, thus ideal for documenting many system requirements. They are arguably easier to comprehend and to maintain than conventional mathematical expressions. Tabular expressions have well-defined formal semantics (e. g., [7]). For our purpose of capturing the input-output requirements of timing function blocks, the tabular structure in Fig. 2 below suffices: rows in the first column denote input conditions, and rows in the second column denote the corresponding output results. The input column may be sub-divided to specify sub-conditions. When the output column denotes a state variable, we may write *NC* to abbreviate the case of "no change" on its value.

In documenting input-output behaviours using horizontal condition tables (HCTs), we need to reason about their *completeness* and *disjointness*. Suppose there is no sub-condition, completeness ensures that

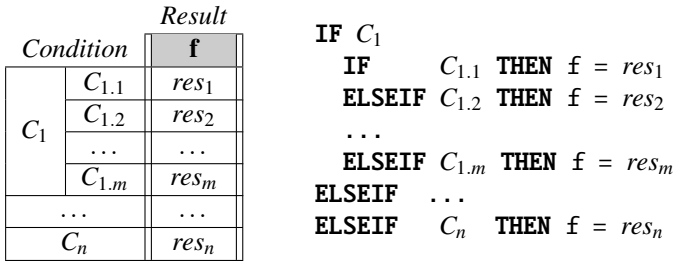


Figure 2: Semantics of Horizontal Condition Table (HCT)

at least one row is applicable to every input, i. e., $(C_1 \vee C_2 \vee \cdots \vee C_n \equiv True)$. Disjointness ensures that rows do not overlap, e. g., $(i \neq j \Rightarrow \neg(C_i \wedge C_j))$. Similar constraints apply to the sub-conditions, if any.

Choice of Theorem Prover. We chose the PVS theorem prover to formalize the input-output requirements of function blocks primarily because it supports the syntax and semantics of tables. In particular, for each table that is syntactically valid, PVS automatically generates its associated healthiness conditions of completeness and disjointness as type correctness conditions (TCCs). Furthermore, we have expertise built from past experience in applying PVS to check requirements and designs in the nuclear domain [8] that gave us confidence in using the toolset. For modelling real-time behaviour, we reused parts of the PVS theories from [5, 4] (see Sec. 2.3 to 2.5).

For presentation, we show PVS listings using ASCII characters in frame boxes, whereas in the main text, we typeset names of predicates, types, theorems, *etc*., in the math form.

2.2 Modelling Time in the Physical Domain

As PLCs are widely used in real-time systems, the modelling of time is critical. For our purpose of verification, we approximate the continuous time in the physical domain as a type *tick*, defined as a discrete series of equally-distributed clock ticks, with an arbitrarily small positive time interval δ between two consecutive clock ticks: $tick = \{t_n : \mathbb{R}_{\geq 0} \mid \delta \in \mathbb{R}_{>0} \wedge (\exists n : \mathbb{N} \bullet t_n = n \times \delta)\}$. We also define *not_init*, a subtype of *tick* that excludes t_0. We define operators to manipulate values at the tick level: $init(t : tick) = (t = 0)$, $pre(t : not\_init) = (t - \delta)$, $next(t : tick) = (t + \delta)$, and $rank(t : tick) = \frac{t}{\delta}$. We often apply induction to prove properties that should hold over time[1]:

```
time_induction: THEOREM
  FORALL (P: pred[tick]):
     (FORALL (t: tick): init(t) => P(t))
   & (FORALL (t: not_init): P(pre(t)) => P(t)) => (FORALL (t: tick): P(t))
```

where $pred[tick]$ is a PVS shorthand for "predicates on tick" (i.e., functions mapping *tick* to Boolean).

2.3 Modelling Samples in the Software Domain

We use a variable $Sample : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ to denote the series of samples over time, such that the time of each sample (i.e., *Sample(n)*, $n \in \mathbb{N}$) maps to a valid clock tick. As shown in Fig. 3, realistically, the clock tick frequency $\frac{1}{\delta}$ in the physical domain should be significantly larger than the sampling frequency in the software domain. We bound sample intervals between *Tmin* and *Tmax*, determined by considering the shortest time after which events must be detected.

As rates of clock ticks and sampling are distinct, a monitored signal *Pf* that rapidly changes between two consecutive samples (called a "spike") can cause inconsistent results produced in the two domains.

[1] `pred[tick]` is synonymous to the function type `[tick -> bool]`

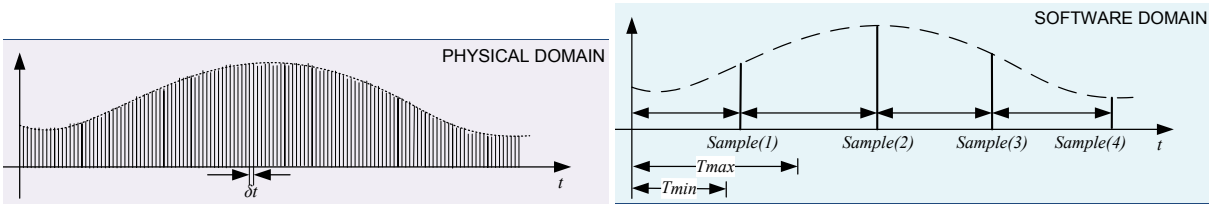


Figure 3: Clock Ticks in Physical Domain vs. Samples in Software Domain [4, p81]

To rule out such scenarios, we define a predicate subtype *FilteredTickPred*[2] that only allows monitored conditions which remain unchanged between consecutive samples:

```
FilteredTickPred?(P: pred[tick]): bool =
      ( FORALL t0: P(t0) /= P(next(t0)) =>
          (FORALL (t: tick):
             t0 < t AND t <= t0 + Tmax => P(next(t0)) = P(t)) )
  AND ( FORALL (t: tick):
           t <= Tmax => P(t) = P(0) )

FilteredTickPred: TYPE+ = (FilteredTickPred?)
Pf: VAR FilteredTickPred
```

2.4 Operators for Specifying Timing Requirements As discussed in Sec. 1, we define the infix operator:

$$Held\_For : (tick \to \mathbb{B}) \times (tick \to \mathbb{R}_{>0}) \times (tick \to \mathbb{R}_{\geq 0}) \times (tick \to \mathbb{R}_{\geq 0}) \to (tick \to bool)$$

to specify a common functional timing requirement, e.g., P **Held_For** $(d, \delta L, \delta R)$, that a monitored boolean condition P should sustain over a positive time duration d, with non-negative left tolerance δL and right tolerance δR. More precisely,

$$P \textbf{ Held\_For } (d, \delta L, \delta R)(t_{now}) \equiv (\ \exists t_j : t_{now} - t_j \geq d \bullet (\forall t_i : t_j \leq t_i \leq t_{now} \bullet P(t_i))\ )$$

where $d \in [d(t_{now}) - \delta L(t_{now}), d(t_{now}) + \delta L(t_{now})]$. In our model of time, inputs and outputs are represented as functions mapping ticks to values. For example, the left tolerance may change from $\delta L(t_1)$ to $\delta L(t_2)$. However, as discussed in Sec. 1, the behaviour of *Held_For* is nondeterministic when P has held *TRUE* for a period that is bounded by $[d - \delta L, d + \delta R]$.

To resolve the non-determinism in *Held_For*, we define two refinement operators: *Held_For_S* and *Held_For_I*. Both operators are deterministic by fixing the duration d in the above definition of *Held_For* as $d(t_{now}) - \delta L(t_{now})$. We will only see *Held_For_I* in the case studies, but it is defined in terms of *Held_For_S*. *Held_For_S* is a partial function on *tick* that produces values only at points of sampling (i.e., it is undefined on ticks in-between samples).

```
Held_For_S(P, duration, Sample)(ne): bool =
  EXISTS (n0 : nat):
        Sample(ne) - Sample(n0) >= duration
    AND FORALL (n: nat): n0 <= n AND n <= ne => P(Sample(n))
```

[2] An example of using the subtype *FilteredTickPred* to constrain input signals can be found in the verification story of the *Pushbutton* subsystem (Sec. 5).

On the other hand, *Held_For_I* is a totalized version of *Held_For_S*: its value at time *t*, where *Sample(n)* $\leq$ *t* $<$ *Sample(n + 1)*, is equivalent to that produced at time *Sample(n)* (i.e., the closest left sample calculated by *Left_Sample*).

```
Held_For_I(P, duration, Sample)(t): bool =
  Held_For_S(P, duration, Sample)(Left_Sample(Sample, t))
```

2.5 Implementing the Held_For_I Timing Operator We use *Timer_I* (defined in terms of *Timer_S*) to implement the *Held_For_I* timing operator. *Timer_I* agrees on outputs from *Timer_S* at sample points and keep the same value at any clock tick until the next sample point (this is analogous to how *Held_For_I* is related to *Held_For_S*).

```
Timer_I(P, Sample, TimeOut)(t): tick =
  Timer_S(P, Sample, TimeOut)(Left_Sample(Sample, t))
```

where *Timer_S*[4, p. 97] counts, starting from the closest left sample to the clock tick in question, for how long the monitored condition *P* has been enabled, and stops counting when *TimeOut* is reached. The output type of *Timer_S* is *tick*, calculated from how many samples *P* has been held across. As mentioned in Sec. 1, the theorem *TimerGeneral_I* is proved [4, p. 99] to ensure that *Timer_I* is a proper implementation for *Held_For_I*.

2.6 A Formal Approach to Specifying and Verifying Function Blocks Our reported approach [10] fits into the timing model as described above. For each FB, its input-output requirements and FBD implementation are formalized in PVS as two (higher-order) predicates, parameterized by input and output lists. Each input or output is represented as a timed sequence (or trajectory) mapping clock ticks to values (e.g., $[tick \rightarrow real]$). Without loss of generality we write *i* and *o* to denote, respectively, the lists of input and output trajectories.

Consider a composite function block *FB* (e.g., see Fig. 9 in Sec. 4). The *requirements predicate* of *FB* (denoted as *FB_REQ*, e.g., *Trip_sealedin_REQ*) returns true if its outputs are related to inputs in the expected way (specified using tabular expressions) across all time ticks. The *implementation predicate* of *FB* (denoted as *FB_IMPL*, e.g., *Trip_sealedin_IMPL*) is constructed by composing, using logical conjunction, the requirements predicates of its component FBs (e.g., *TON*, *CONJU*, *etc.*) as configured in its FBD implementation. All inter-connectives (e.g., *w1*, *w2*, *etc.*) in the FBD implementation are hidden using an existential quantification.

Proof of Consistency To ensure that the implementation is consistent or feasible, we prove that for each list of input trajectories, there exists at least one list of output trajectories such that *FB_IMPL* is defined:

$$\vdash \forall i \bullet \exists o \bullet FB\_IMPL(i, o) \tag{2}$$

Proof of Correctness To ensure that the implementation is correct with respect to the intended requirement, we prove that the observable inputs and outputs conform to those of the requirements:

$$\vdash \forall i \bullet \forall o \bullet FB\_IMPL(i, o) \Rightarrow FB\_REQ(i, o) \tag{3}$$

3 Formalizing IEC 61131-3 Timers with Tolerances

We present the first contribution of this paper: incorporating the notion of *timing tolerances* [15] (i.e., the controller's reaction to the environment is associated with a delay) into the formalization of the black-box, input-output requirements of IEC 61131-3 timers. Such formalization improves the accuracy of our previous work [10] by making the resulting requirements models *implementable*.

In IEC 61131-3 there are three timer FBs: *TON* (On-delay), *TOF* (Off-delay), and *TP* (Pulse) timers. As case studies presented in this paper (Sec. 4 and Sec.5) only make use of the *TON* block, in this section we present its re-formalization only and report details of the other two timer blocks in [11].

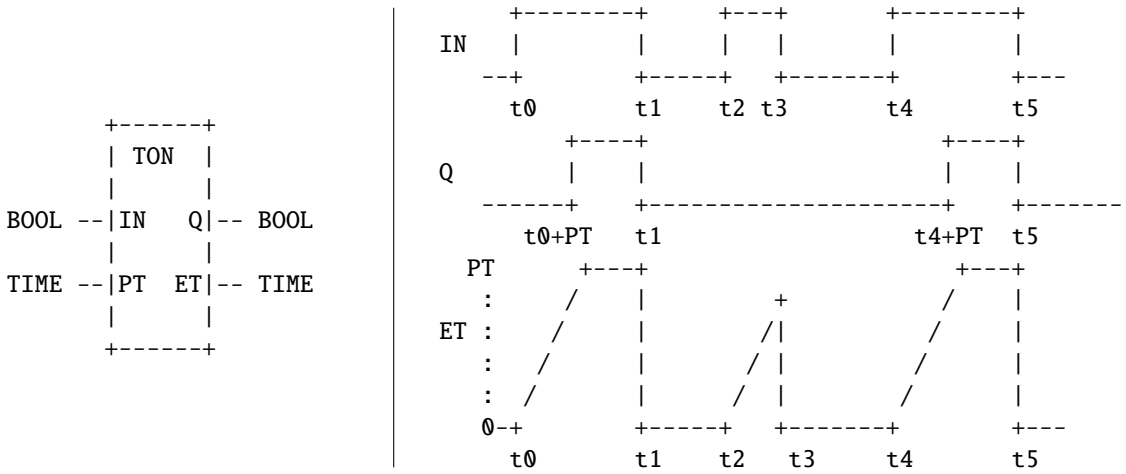


Figure 4: *TON* timer declaration and definition in timing diagram [6]

The *TON* block is commonly used as a component of safety-critical systems. For example, it can be used to determine if a sensor signal has gone out of its safety range for too long, as we will see in Sec. 4 and Sec. 5. Fig. 4 shows, extracted from IEC 61131-3, the input-output declaration (on the LHS) and a timing diagram[3] (on the RHS) illustrating the expected behaviour of the *TON* block. The *TON* block is declared with two inputs (a boolean condition *IN* and a time period of length *PT*) and two outputs (a boolean value *Q* and a length *ET* of time period). Timer *TON* monitors the input condition *IN* and sets the output *Q* as true whenever *IN* remains enabled for longer than a time period of some input length *PT*. If the monitored input *IN* has been enabled for some time $t < PT$, then the timer sets the output *ET* (i.e., elapsed time) with value *t*; otherwise, it sets *ET* with value *PT*.

The use of a timing diagram by IEC 61131-3 to describe the expected behaviour of the *TON* block (and the other two timers) is limited to an incomplete set of use cases. As a result, we attempted in [10] to use function tables to formalize the black-box, input-output requirements of the three timer blocks (on-delay, off-delay, and pulse timers) listed in IEC 61131-3. Fig. 5 shows our previous attempt of the requirements specification of the *TON* block, where *t* denotes the current clock tick, and a time stamp *last_enabled* is used to record the exact time (with no delay) that the input condition *IN* just becomes enabled. However, the requirements model in Fig. 5 is not implementable because it describes idealized behaviour: the timer (or the controller) reacts instantaneously to changes in the environment.

As part of the contribution of this paper, we revise the function tables of all three timers in IEC 61131-3 by incorporating the notion of timing tolerances [15]. To achieve this, we use the pre-verified operator *Timer_I* (Sec. 2) to redefine requirements of the three timers (e.g., Fig. 6 for the *TON* timer).

The essence of our first contribution presented in this section is that we incorporate the notion of timing tolerances, via the use of the pre-verified operator *Timer_I*, into the requirements of IEC 61131 timers so that they are implementable. This allows us to conduct case studies such as the one in Sec. 4 on implementing and verifying subsystems using the IEC 61131-3 timers.

[3]The horizontal axis is labelled with time instants t_i, $i \in 0..5$

Condition	*Result* last_enabled
$\neg IN_{-1} \wedge IN$	t
$IN_{-1} \vee \neg IN$	NC

Condition	*Result* Q
$IN \wedge (d \geq PT)$	TRUE
$IN \wedge (d < PT)$	FALSE
$\neg IN$	FALSE

Condition	*Result* ET
$IN \wedge (d \geq PT)$	PT
$IN \wedge (d < PT)$	d
$\neg IN$	0

where d stands for duration, d = t - last_enabled

Figure 5: Tabular Requirements of Timer *TON*: Idealized Behaviour

Condition	*Result* Q
$d \geq PT$	TRUE
$d < PT$	FALSE

Condition	*Result* ET
$d \geq PT$	PT
$d < PT$	d
$\neg IN$	0

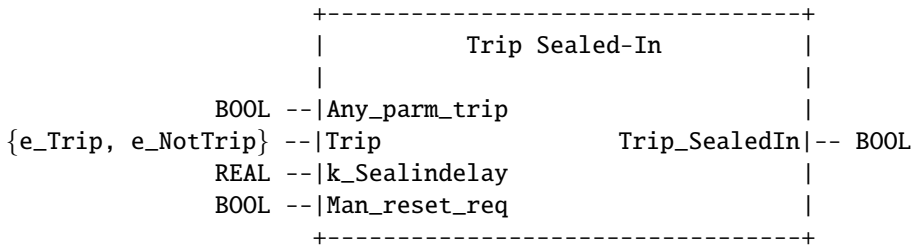
where d stands for duration, d = (IN) Timer_I (PT, δL, δR)

Figure 6: Tabular Requirements of Timer *TON*: Timing Tolerances Incorporated

4 Case Study 1: the *Trip Sealed-In* Subsystem

In this section we apply our approach (Sec. 2.6) to verify a candidate FBD implementation for the *Trip Sealed-In* subsystem. We identify an initialization error and suggest a fix.

4.1 Input-Output Declaration and Informal Description The figure below declares the inputs and outputs of the *Trip Sealed-In* subsystem:



Trip Sealed-In is a generic subsystem which monitors: 1) a set of sensor values; and 2) an alarm value produced by some other subsystem. It signals an alarm (denoted by the output *Trip_SealedIn*), which may be manipulated by other subsystems, when two conditions are met. First, any of the monitored sensor values goes out of its safety range (called a parameter trip and denoted by an input condition *Any_parm_trip*). Second, the monitored input alarm is signalled continuously for longer than some preset constant *k_Sealindelay*[4] amount of time (denoted by an input value *Trip* of enumerated type $\{e\_Trip, e\_NotTrip\}$). Once the alarm *Trip_SealedIn* is activated, it is not deactivated until all monitored sensor values fall back within their safety ranges, and then a manual reset is requested (denoted as an input *Man_reset_req*).

4.2 Tabular Requirements Specification with Timing Tolerances We use a function table (Fig. 7) to perform a complete and disjoint analysis on the input domains. To incorporate timing tolerances into the requirements of *Trip Sealed-In*, we use the non-deterministic *Held_For* operator (Sec. 2) to specify a sustained window of duration $[k\_Sealindelay - \delta L, k\_Sealindelay + \delta R]$.

[4]The $k\_$ name prefix is reserved for system-wide constants.

Condition		*Result* *Trip_SealedIn*
Any_parm_trip	(*Trip*=*e_Trip*) **Held_For** (*k_Sealindelay*, δL, δR)	TRUE
	¬[(*Trip*=*e_Trip*) **Held_For** (*k_Sealindelay*, δL, δR)]	NC
¬*Any_parm_trip*	*Man_reset_req*	FALSE
	¬*Man_reset_req*	NC

Figure 7: *Trip Sealed-In*: (non-deterministic) Requirements of with Tolerances

However, for the purpose of verification in PVS, we reformulate the non-deterministic behaviour of Fig. 7 in a recursive function[5] using the deterministic *Held_For_I* operator to impose the constraint that only a single value (i.e., *k_Sealindelay* − *delta_L* where both are declared constants) is chosen from the duration and is used consistently for detecting sustained events.

Below we define a recursive function *Trip_SealedIn_f* over all clock ticks:

```
Trip_SealedIn_f(Any_parm_trip: pred[tick],
                Trip         : [tick->{e_Trip, e_NotTrip}],
                Man_reset_req: pred[tick])(t: tick)
: RECURSIVE bool =
  IF init(t) THEN TRUE ELSE
  LET
   TRIPPED = LAMBDA (t: tick): Trip(t) = e_Trip,
   HELD    = Held_For_I(TRIPPED,k_Sealindelay-delta_L,Sample)(t),
   PREV    = Trip_SealedIn_f(
                Any_parm_trip,Trip,Man_reset_req)(pre(t))
  IN TABLE
     %-------------------------------------------------------%
     |     Any_parm_trip(t) &     HELD             | TRUE    ||
     %-------------------------------------------------------%
     |     Any_parm_trip(t) & NOT HELD             | PREV    ||
     %-------------------------------------------------------%
     | NOT Any_parm_trip(t) &     Man_reset_req(t) | FALSE   ||
     %-------------------------------------------------------%
     | NOT Any_parm_trip(t) & NOT Man_reset_req(t) | PREV    ||
     %-------------------------------------------------------%
  ENDTABLE ENDIF MEASURE rank(t)
```

Using *Trip_SealedIn_f*, we have deterministic requirements (Fig. 8) for the *Trip Sealed-In* subsystem:

Remark. Compared with Fig. 7, the use of the operator *Held_For_I* in Fig. 8 resolves the non-determinism by fixing the level of tolerance (i.e., as the alarm input *Trip* has been activated for or longer than *k_Sealindelay* − δL, the *Trip Sealed-In* subsystem is guaranteed to detect it and act accordingly).

4.3 Formalizing the FBD Implementation We propose a FBD implementation (Fig. 9) which should satisfy the requirements (Fig. 8).

We use the IEC 61131 *TON* timer (see Sec. 3 for its formalization incorporated with tolerances) to implement the use of the *Held_For_I* operator (subject to a correctness proof which we will discuss below). As the recursive function used to define the requirements depends on the value of itself (at the previous time tick), we specify a feedback loop (dashed line) in the implementation.

[5]For proving termination, its progress is measured using discrete time instants *rank(t)*.

```
Trip_SealedIn_REQ(Any_parm_trip: pred[tick],
                  Trip         : [tick->{e_Trip, e_NotTrip}],
                  Man_reset_req: pred[tick],
                  TripSealedIn : pred[tick]): bool
= FORALL (t: tick):
    TripSealedIn(t) =
      Trip_SealedIn_f(Any_parm_trip, Trip, Man_reset_req)(t)
```

Figure 8: *Trip Sealed-In*: (deterministic) Requirements of with Tolerances in PVS

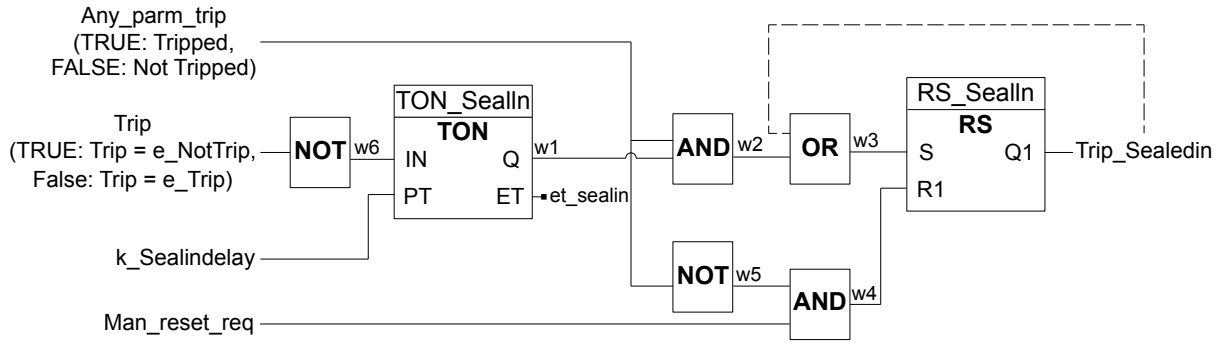


Figure 9: *Trip Sealed-In* implementation in FBD

The use of the left-most *NOT* (negation) block in Fig. 9 has to do with the mismatch between types at the requirements level (i.e., $\{e\_Trip, e\_NotTrip\}$) and that at the FB implementation level (i.e., boolean): somehow the engineers interpret value *e_Trip* as *FALSE* and *e_NotTrip* as *TRUE*, so a conversion is necessary to make sure the *Trip Sealed-In* has a consistent interpretation. The requirements that the alarm output *Trip_Sealedin* is deactivated (or reset) when there is no parameter trips, and when a manual reset is requested, is implemented using a standard block *RS* (reset dominant flip flop).

To prove that the proposed FBD implementation of *Trip Sealed-In* (Fig. 9) is both feasible and conforms to its requirements (Fig. 8), we follow our approach (Sec. 2.6) to formalize it by composing, using conjunction, the formalizing predicates[6] of all component blocks (all inter-connectors are hidden using an existential quantification.):

$$
\begin{aligned}
&Trip\_sealedin\_IMPL(Any\_parm\_trip, Trip, Man\_reset\_req, Trip\_SealedIn) \\
&\quad \equiv \exists\, w_1, w_2, w_3, w_4, w_5, w_6, et\_sealin \bullet \\
&\qquad \left(\begin{array}{l}
\quad NOT(Trip, w_6) \\
\wedge\, TON(w_6, k\_Sealindelay - \delta L, w_1, et\_sealin) \\
\wedge\, CONJ(Any\_parm\_trip, w_1, w_2) \\
\wedge\, DISJ(w_2, Trip\_SealedIn, w_3) \\
\wedge\, NOT(Any\_parm\_trip, w_5) \\
\wedge\, CONJ(w_5, Man\_reset\_req, w_4) \\
\wedge\, RS(w_4, w_3, Trip\_SealedIn)
\end{array} \right)
\end{aligned}
$$

[6]Predicates *NOT* (logical negation), *CONJ* (logical conjunction), *DISJ* (logical disjunction), *TON* (on-delay timer), and *RS* (reset dominant latch).

4.4 Proofs of Consistency and Correctness First, we prove that the FBD implementation (Fig. 9) is feasible by instantiating formula (2) in Sec. 2.6:

$$\begin{array}{l}\vdash \forall\, \mathit{Any\_parm\_trip}, \mathit{Trip}, \mathit{Man\_reset\_req} \bullet \\ \quad \exists\, \mathit{Trip\_SealedIn} \bullet \mathit{Trip\_sealedin\_IMPL}(\\ \qquad \mathit{Any\_parm\_trip},\ \mathit{AbstParmTrip\_timed(Trip)},\ \mathit{Man\_reset\_req},\ \mathit{Trip\_SealedIn})\end{array}$$

The abstraction function *AbstParmTrip_timed* handles the mismatched types of input *Trip* at levels of requirements and implementation (e.g., *e_NotTrip* mapped to *TRUE*). We discharge the consistency proof using proper instantiations.

Second, we prove that the FBD implementation is correct with respect to Fig. 8, considering timing tolerances, by instantiating formula (3) in Sec. 2.6:

$$\begin{array}{l}\vdash \forall\, \mathit{Any\_parm\_trip},\ \mathit{Trip},\ \mathit{Man\_reset\_req},\ \mathit{Trip\_SealedIn} \bullet \\ \quad \mathit{Trip\_sealedin\_IMPL}(\mathit{Any\_parm\_trip}, \mathit{AbstParmTrip\_timed(Trip)}, \mathit{Man\_reset\_req}, \mathit{Trip\_SealedIn}) \\ \qquad \Rightarrow \mathit{Trip\_sealedin\_REQ}(\mathit{Any\_parm\_trip}, \mathit{Trip}, \mathit{Man\_reset\_req}, \mathit{Trip\_SealedIn})\end{array}$$

As there is a feedback loop in the FBD implementation (Fig. 9), our strategy of discharging the correctness theorem is by mathematical induction (using the *time_induction* proposition in Sec. 2) over tick values. Since the *Timer_I* operator (Sec. 2) is used to formalize the requirements of the *TON* timer that contributes to the FBD implementation, its definition is expanded in both the base and inductive cases.

However, when proving the base case (when $t = 0$), we found that the initial value of output *Q1* of the *RS_Sealin* block and the initial value of the subsystem output *Trip_SealedIn* — these two values are directly connected in the initial FBD implementation (Fig. 9) — are inconsistent. According to the SRS (Software Requirements Specification), the value of *Trip_SealedIn* is initialized to *TRUE*, whereas that of *Q1* is *FALSE*. We resolve this issue of inconsistency by suggesting a revised FBD implementation (Fig. 10) and prove that it is correct with respect to Fig. 8. In this revised implementation, we add an IEC 61131-3 selection block *SEL_Sealin*, acting as a multiplexer to discriminate the value of *Q1* (at the initial tick and at the non-initial tick) that is output as *Trip_SealedIn*.

Remark. We just illustrated that, by adopting our approach, we are able to justify the appropriateness of a candidate FBD implementation, and to fix it accordingly if necessary.

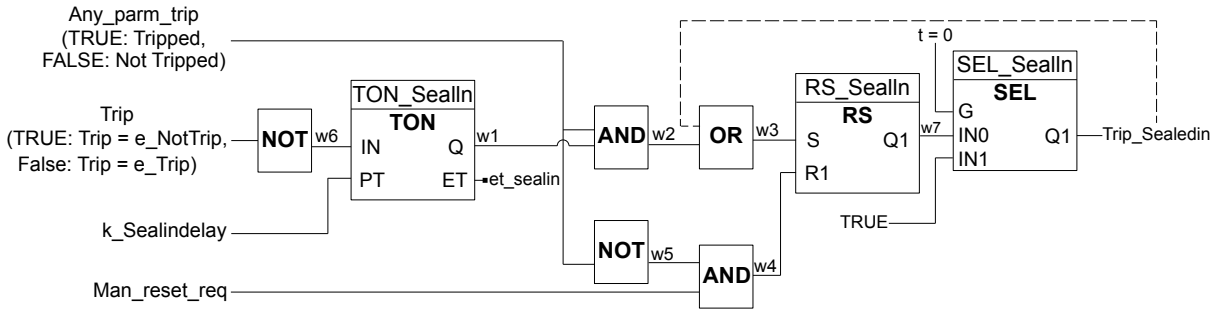


Figure 10: Revised *Trip Sealed-In* implementation in FBD

5 Case Study 2: the *Pushbutton* Subsystem

In this section we apply our approach (Sec. 2.6) to verify a candidate FBD implementation for the *Pushbutton* subsystem. We identify a missing assumption of implementation and suggest a solution.

5.1 Input-Output Declaration and Informal Description The figure below declares the inputs and outputs of the *Pushbutton* subsystem.

```
        +-----------------------------------+
        |            Pushbutton             |
        |                                   |
y_pb --|m                                   |
REAL --|k_Debounce              f_Pushbutton|-- y_pbdesign
REAL --|k_Stuck                             |
        +-----------------------------------+
```

Pushbutton is a generic subsystem which monitors the status of a pushbutton (denoted by an input $m \in \{e\_Pressed, e\_NotPressed\}$), which may be pressed to manually, e.g., enable or disable a sensor trip[7]. Its behaviour is denoted by an output $f\_Pushbutton \in \{e\_pbNotDebounced, e\_pbDebounced, e\_pbStuck\}$. *Pushbutton* determines if either: (a) the button is not pressed, or pressed but not for a sufficient period of time (denoted by some pre-set value $k\_Debounce$[8]) to register as a press; (b) the button is pressed long enough to quality as a press; or (c) the button is pressed for longer than some pre-set period of time (denoted by k_Stuck) without bouncing back and thus is considered stuck.

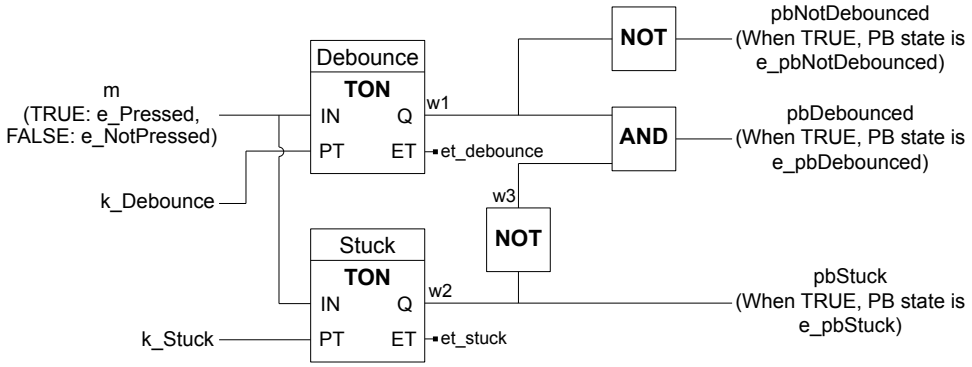
5.2 Tabular Requirements Specification with Timing Tolerances For the purpose of verification in PVS, we use the function table below[9] to perform a complete and disjoint analysis on the domain of the button status. To incorporate timing tolerances, similar to the requirements specification for the *Trip Sealed-In* subsystem (Fig. 8, p.74), we use the deterministic *Held_For_I* operator (Sec. 2), where values $k\_Debounce - \delta L$ and $k\_Stuck - \delta L$ are chosen and used consistently for detecting the sustained events.

	Result
Condition	$f\_Pushbutton$
$m = e\_NotPressed$	$e\_pbNotDebounced$
$(m = e\_Pressed) \wedge \neg debounced$	$e\_pbNotDebounced$
$debounced \wedge \neg stuck$	$e\_pbDebounced$
$stuck$	$e\_pbStuck$

where $debounced = (m = e\_Pressed)$ **Held_For_I** $(k\_Debounce - \delta L)$

$stuck = (m = e\_Pressed)$ **Held_For_I** $(k\_Stuck - \delta L)$

5.3 Formalizing the FBD Implementation We propose a FBD implementation which should satisfy the requirements:



[7] A sensor trip occurs if the sensor signal in question goes above its set point.

[8] The $k\_$ name prefix is reserved for system-wide constants.

[9] The PVS encoding of this table is not shown in this paper.

We use two IEC 61131 *TON* timers (see Sec. 3 for its formalization) to implement the predicates *debounced* and *stuck* in the above requirements table that involve the use of the *Held_For_I* operator. Since only the button status is monitored, there is no need to specify a feedback loop in the implementation. To prove that this FBD implementation is consistent and correct, similar to what we do for that for the *Trip Sealed-In* subsystem (see Fig. 9, p.74), we formalize it by composing the formalizing predicates of all its component blocks using conjunction, and by hiding inter-connectors using an existential quantification.

5.4 Proof Obligations: Consistency and Correctness The consistency and correctness theorems for the *Pushbutton* subsystem are stated in a similar manner as those for the *Trip Sealed-In* subsystem by properly instantiating, respectively, formulas 2 and 3 in Sec. 2.6. However, we had difficulties when first attempting to prove that the above requirements table for *f_Pushbutton* possesses the disjointness property. To resolve this, we tried to simplify the requirements table by collapsing the first two rows into a single one with the input condition $\neg pressed \wedge \neg stuck$. This is done based on the observations that both row 1 and row 2 map to the same output value *e_pbNotDebounced*, and that $m = e\_Pressed \vee m = e\_NotPressed \equiv true$.

When proving that the revised requirements table is equivalent to the original one, we found a problematic scenario where the value of output *f_Pushbutton* is produced inconsistently at the requirements and implementation levels: when the input condition *m* varies rapidly and generates a "spike", whose duration is shorter than the timing resolution. To rule out the "spike" scenarios for input *m*, we added an assumption, at the FBD implementation level, using the predicate subtype *FilteredTickPred* (Sec. 2).

Finally, the revised requirements table can be proved for its completeness, disjointness, consistency, and correctness by following a similar pattern of proofs as for the *Trip Sealed-In* subsystem. For proving the correctness theorem, as there is not a feedback loop in the above FBD implementation, we do not need to discharge the correctness theorem using mathematical induction. Furthermore, as the *TON* components in the FBD implementation are formalized using the *Timer_I* operator (Sec. 3), we need to reuse the theorem *TimerGeneral_I* with proper instantiations to show their equivalence to the *Held_For_I* expressions in the revised requirements table.

6 Proof Structure

In the industrial software control system that we consider for this paper, the *Trip Sealed-In* subsystem implemented using a feedback loop (Sec. 4) and the *Pushbutton* subsystem (Sec. 5) are representative[10] of functionality in which FBD implementations make use of IEC 61131 timer blocks. Structures of their consistency and correctness proofs shall guide the proofs for many other subsystems of a similar nature.

For illustration, we consider the correctness proof structure for the *Trip Sealed-In* subsystem. In principle, there are eight key steps to discharge the correctness theorem for a real-time subsystem implemented with a feedback loop (e.g., *Trip Sealed-In*). Except for the fourth step, where the *time_induction* theorem is used to handle the feedback loop, others are standard commands.

1) Apply `skosimp` to eliminate the universal quantification over input and output variables, and then apply `flatten` to simplify theorem structure $impl \Rightarrow req$ by moving *impl* to the antecedent and *req* to the consequent. 2) Apply multiple `expand` commands to unfold definitions of the requirements and implementation predicates. 3) In the antecedent, apply `skolem!` to eliminate the existential quantification over inter-connectors. 4) To handle the recursive feedback loop, use the theorem *time_induction* on $t \in tick$.

[10] This judgement is based on the use of a generic timing function, the *Held_For* operator, in the tabular expressions that describe the required behaviour.

5) Apply a series of basic commands to complete the proof for the base case. 6) To prove the inductive case, first apply `skolem!` and then `expand` to unfold the recursive function that is used to define the requirements predicate (e.g., see Fig. 8, p.74). 7) Apply `split` and `lift-if` to generate sub-goals. 8) Repeatedly apply: `expand` commands to unfold definitions of the predicates for internal components, theorem *TimerGeneral_I* with proper instantiations to link between *Held_For_I* in the requirements and *Timer_I* in the implementation, and basic commands to complete the proof for the inductive step.

7 Related Work

The focus of this paper is the practical verification of real-time behaviour against timing requirements with tolerances. Our approach to specifying and verifying FBs [10], compared with others on verifying PLC programs in contexts of model checking and theorem proving, is novel in three aspects: (1) extent of the case study; (2) practical application in the safety-critical industry; and (3) mature tool support of theorem proving.

In our formal setting, proving that an FBD implementation is correct (with respect to its intended input-output timing requirements) is essentially proving that it is a valid refinement. However, our purpose of verification is on the observable input-output behaviour, as opposed other properties such as boundedness, liveness, and robustness (e.g., [3, 16, 13, 2]). Of more relevance is the use of timed automata to model timing tolerances with ASAP (as soon as possible) semantics to verify the correctness of implementation [17], but with no suggestion for either tool support or its adoption in practice.

8 Conclusion

In this paper we report our application of a formal approach on using FBs (including timers) from IEC 61131-3 to verify two subsystems of an industrial software control system from the nuclear domain. We re-formalize all three IEC 61131-3 timers to incorporate the notion of tolerances. Specifically, we use the re-formalized IEC 61131 on-delay timer for the proposed FBD implementations, and prove that they are feasible and correct (i.e., satisfies the intended timing requirements). While attempting to verify the two subsystems, we find an issue of initialization failure, and an issue of missing implementation assumption. In both cases, we suggest possible solutions. We identify patterns of proof commands that are amenable to strategies that will facilitate the automated verification of the feasibility and correctness of other subsystems. As ongoing and future work, we first aim to verify subsystems with more sophisticated timing requirements, e.g., nested *Held_For* expressions. Second, we aim to prove safety properties from the composition of real-time subsystems. Third, we aim to automate the process of proofs that share a common structure.

References

[1] (2011): *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. Special Committee 205 of RTCA.

[2] Ed Brinksma, Angelika Mader & Ansgar Fehnker (2002): *Verification and optimization of a PLC control schedule*. *International Journal on Software Tools for Technology Transfer (STTT)* 4(1), pp. 21–33. Available at `http://dx.doi.org/10.1007/s10009-002-0079-0`.

[3] Zhijun Ding, Changjun Jiang & Mengchu Zhou (2013): *Design, Analysis and Verification of Real-Time Systems Based on Time Petri Net Refinement*. *ACM Trans. Embed. Comput. Syst.* 12(1), pp. 4:1–4:18. Available at http://dx.doi.org/10.1145/2406336.2406340.

[4] Xiayong Hu (2008): *Proving implementability of timing properties with tolerance*. Ph.D. thesis, McMaster University, Department of Computing and Software.

[5] Xiayong Hu, Mark Lawford & Alan Wassyng (2009): *Formal Verification of the Implementability of Timing Requirements*. In: *FMICS*, *LNCS* 5596, Springer, pp. 119–134. Available at http://dx.doi.org/10.1007/978-3-642-03240-0_12.

[6] IEC (2003): *61131-3 Ed. 2.0 en:2003: Programmable Controllers — Part 3: Programming Languages*. International Electrotechnical Commission.

[7] Ying Jin & David Lorge Parnas (2010): *Defining The Meaning of Tabular Mathematical Expressions*. *Science of Computer Programming* 75(11), pp. 980 – 1000. Available at http://dx.doi.org/10.1016/j.scico.2009.12.009.

[8] Mark Lawford, Jeff McDougall, Peter Froebel & Greg Moum (2000): *Practical application of functional and relational methods for the specification and verification of safety critical software*. In: *Proc. of AMAST 2000*, *LNCS* 1816, Springer, pp. 73–88. Available at http://dx.doi.org/10.1007/3-540-45499-3_8.

[9] Sam Owre, John M. Rushby & Natarajan Shankar (1992): *PVS: A Prototype Verification System*. In: *CADE*, *LNCS* 607, pp. 748–752. Available at http://dx.doi.org/10.1007/3-540-55602-8_217.

[10] Linna Pang, Chen-Wei Wang, Mark Lawford & Alan Wassyng (2013): *Formalizing and Verifying Function Blocks using Tabular Expressions and PVS*. In: *FTSCS*, *Communications in Computer and Information Science* 419, Spring, pp. 163–178. Available at http://dx.doi.org/10.1007/978-3-319-05416-2_9.

[11] Linna Pang, Chen-Wei Wang, Mark Lawford, Alan Wassyng, Josh Newell, Vera Chow & David Tremaine (2014): *Formal Verification of Real-Time Function Blocks using PVS*. Technical Report 16, McSCert. https://www.mcscert.ca/index.php/documents/mcscert-reports?view=publication&task=show&id=16.

[12] David Lorge Parnas, Jan Madey & Michal Iglewski (1994): *Precise Documentation of Well-Structured Programs*. *IEEE Transactions on Software Engineering* 20, pp. 948–976. Available at http://dx.doi.org/10.1109/32.368133.

[13] Ocan Sankur (2013): *Shrinktech: A Tool for the Robustness Analysis of Timed Automata*. In: *Computer Aided Verification*, *LNCS* 8044, Springer, pp. 1006–1012. Available at http://dx.doi.org/10.1007/978-3-642-39799-8_72.

[14] Alan Wassyng & Mark Lawford (2003): *Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project*. In: *FME 2003*, *LNCS* 2805, Springer, pp. 133–153. Available at http://dx.doi.org/10.1007/978-3-540-45236-2_9.

[15] Alan Wassyng, Mark Lawford & Xiaoyong Hu (2005): *Timing Tolerances in Safety-Critical Software*. In: *FM 2005*, *LNCS* 3582, Springer, pp. 157 – 172. Available at http://dx.doi.org/10.1007/11526841_12.

[16] Anton Wijs & Luc Engelen (2013): *Efficient Property Preservation Checking of Model Refinements*. In: *TACAS*, *LNCS* 7795, Springer, pp. 565–579. Available at http://dx.doi.org/10.1007/978-3-642-36742-7_41.

[17] Martin De Wulf, Laurent Doyen & Jean-Franois Raskin (2005): *Almost ASAP semantics: from timed models to timed implementations*. *FAC* 17(3), pp. 319–341. Available at http://dx.doi.org/10.1007/978-3-540-24743-2_20.

Using Indexed and Synchronous Events to Model and Validate Cyber-Physical Systems

Chen-Wei Wang, Jonathan S. Ostroff, and Simon Hudon
Department of Electrical Engineering and Computer Science,
York University, Canada
{jackie, jonathan, simon}@cse.yorku.ca

Timed Transition Models (TTMs) are event-based descriptions for modelling, specifying, and verifying discrete real-time systems. An event can be spontaneous, fair, or timed with specified bounds. TTMs have a textual syntax, an operational semantics, and an automated tool supporting linear-time temporal logic. We extend TTMs and its tool with two novel modelling features for writing high-level specifications: indexed events and synchronous events. Indexed events allow for concise description of behaviour common to a set of actors. The indexing construct allows us to select a specific actor and to specify a temporal property for that actor. We use indexed events to validate the requirements of a train control system. Synchronous events allow developers to decompose simultaneous state updates into actions of separate events. To specify the intended data flow among synchronized actions, we use primed variables to reference the post-state (i.e., one resulted from taking the synchronized actions). The TTM tool automatically infers the data flow from synchronous events, and reports errors on inconsistencies due to circular data flow. We use synchronous events to validate part of the requirements of a nuclear shutdown system. In both case studies, we show how the new notation facilitates the formal validation of system requirements, and use the TTM tool to verify safety, liveness, and real-time properties.

1 Introduction

Cyber-physical systems integrate computational systems (the "controller") with physical processes (the "plant"). Such systems are found in areas as diverse as aerospace, automotive, energy, healthcare, manufacturing, transportation, and consumer appliances. A main challenge in developing cyber-physical systems is modelling the joint dynamics of computer controllers and the plant [1].

Timed Transition Models (TTMs) are event-based descriptions for modelling, specifying, and verifying discrete real-time systems. A system is composed of module instances. Each module declares an interface and a list of events. An event can be spontaneous, fair, or timed (i.e., with lower and upper time bounds). In [6], we provided TTMs with a textual syntax, an operational semantics, and an automated tool, including an editor with type checking, a graphical simulator, and a verifier for linear-time temporal logic. So far, TTMs were used to verify that a variety of implementations satisfy their specifications.

In this paper, we extend the TTM notation, semantics, and tool for two novel modelling features: indexed events and synchronous events. These constructs are suitable for writing high-level specification, and can thus facilitate the validation of system requirements.

Indexed events allow for concise description of behaviour common to a (possibly unspecified) set of actors. The indexing construct allows us to select a specific actor (such as a train) and specify a temporal property for that actor. For example, let *loc* be an array of train locations (a train can be on either the entrance block, a platform, an exit block, or outside the station). An event *move_out* can be indexed with a set *TRAIN* of trains, which results in an indexed event *move_out(t: fair TRAIN)* describing the action

J. Pang, Y. Liu, and S. Mauw (Eds.): 4th International Workshop on Engineering Safety and Security Systems 2015 (ESSS'15)
EPTCS 184, 2015, pp. 81–95, doi:10.4204/EPTCS.184.6

of a train t moving out of a platform and into the exit block. As a result, the event index t can be used to specify the liveness property that every train t waiting at one of the platforms (denoted by the set *PLF*) eventually moves out, and into the exit block: $\Box(loc[t] \in PLF \Rightarrow \Diamond move\_out(t))$. Without the index t, we can only state a weaker property that some train eventually leaves the station (unless we introduce auxiliary variables or events).

Synchronous events allow developers to decompose simultaneous state updates into actions of separate events. However, without a mechanism to reference the post-state values of monitored variables, we cannot properly model the joint actions of the environment and controller. For example, the synchronized action $m := exp \parallel c := f(m)$ specifies that the new (or next-state) value of controlled variable c is computed on the basis of the old (or pre-state) value of monitored variable m (i.e., *exp*). To resolve this, we use primed variables on the RHS of assignments in event actions to denote post-state values. For example, the synchronized action $m := exp \parallel c := f(m')$ specifies that the post-state value of c is now a function on the post-value value of m. Synchronous events, together with primed variables, are suitable for describing high-level specifications used in shutdown systems of nuclear reactors [11]. In such systems, the next-state value of the system controlled variables are expressed in terms of the current-state and next-state values of the monitored variables of nuclear reactors. This allows for a simplified description of the requirements that will later be refined to code.

Contributions. To support indexed and synchronous events for validating requirements, we extend the semantics of TTM (Sec. 2), and we extend our tool accordingly. For synchronous events, our tool automatically infers the data flow, and reports on inconsistencies due to circular data flow. We conduct two realistic case studies: a train control system (Sec. 3) using indexed events, and a part of a nuclear shutdown system (Sec. 4) using synchronous events.

Resources. Complete details of the two case studies are included in an extended report [10], which also contains more case studies of cyber physical systems (i.e., a mutual exclusion protocol, and function blocks from the IEC 61131 Standard for programmable logic controllers) that can be specified using the new notations. Complete TTM listings of the case studies are available at: `https://wiki.eecs.yorku.ca/project/ttm/index_sync_evt`.

2 Semantics for Indexed and Synchronous Events

We extend the one-step operational semantics of TTMs reported in [6] to support both indexed events (Sec. 2.2) and synchronous events (Sec. 2.3). The extensions involve redefining: 1) the abstract syntax of events which affects the rules of transitions and scheduling; and 2) the rules of module compositions. We include the most relevant details to present these extensions, while the complete account of the new semantics is included in an extended report [10, Sec. 6].

2.1 Abstract Syntax: Introducing Fair and Demonic Event Indices We define the abstract syntax of a TTM module instance $\mathscr{M}$ as a 5-tuple (V, s_0, T, t_0, E) where 1) V is a set of local or interface variables; 2) T is a set of timers; 3) E is a set of state-changing events; 4) $s_0 \in$ STATE is the initial state (STATE $\triangleq V \to$ VALUE); and 5) $t_0 \in$ TIMER is the initial timer assignment (TIMER $\triangleq T \to \mathbb{N}$). We define $type \in T \to \mathbb{P}(\mathbb{N})$ and $boundt \in T \to \mathbb{N}$ for querying about, respectively, the type and upper bound of each timer. For example, if timer t_1 is declared as $t_1 : 0..5$, then $boundt(t_1) = 5$ and $type(t_1) = \{0..6\}$. Timers count up to one beyond the specified bound, and remain unchanged until they are started again. The figure below presents the generic form of a TTM event, where $V = \{v_1, v_2, v_3, \cdots\}$ and $T = \{t_1, t_2, t_3, t_4, \cdots\}$.

Concrete syntax of event e:

event_id (x : **fair** T_x; y : T_y) [l,u] **just**
when *grd*
start t_1, t_2
stop t_3, t_4
do $v_1 := exp_1$,
if *condition* **then** $v_2 := v_1' + exp_2$
else skip fi,
v_3 :: 1..4
end

Abstract syntax of the event e:

- $e.id \in$ ID;
- $e.f\_ind \subseteq$ ID ; $e.d\_ind \subseteq$ ID
- $e.d\_ind \triangleq e.f\_ind \cup d\_ind$
- $e.l \in \mathbb{N}$; $e.u \in \mathbb{N} \cup \{\infty\}$
- $e.fair \in \{\text{spontaneous}, \text{just}, \text{compassionate}\}$
- $e.grd \in \text{STATE} \times \text{TIMER} \rightarrow \text{BOOL}$;
- $e.start \subseteq T$;
- $e.stop \subseteq T$;
- $e.action \in \text{STATE} \times \text{TIMER} \leftrightarrow \text{STATE}$;

We use a 10-tuple $(id, f\_ind, d\_ind, l, u, fair, grd, start, stop, action)$ to define the abstract syntax of an event e. We write $e.id$ for its identifier. Sets $e.f\_ind$ and $e.d\_ind$ contain, respectively, fair and demonic indices that can be referenced in the event. Its fairness assumption (i.e., $e.fair$), as discussed in Sec. 2.2, filters out certain execution traces that will be considered in the model checking process. Its guard (i.e., $e.grd$) is a Boolean expression referencing state variables, timers, or its indices. An event e must be taken between its lower time bound (LTB) $e.l$ and upper time bound (UTB) $e.u$, while its guard $e.grd$ remains true. The event action involves simultaneous assignments to $v_1, v_2, \cdots$. We write v_3 :: 1..4 for a demonic (non-deterministic) assignment to v_3 from a finite range. Therefore, its state effect is a relation $e.action$ on state variables and timers. On the RHS of an assignment $y := x$, the state variable x may be "primed" (x') or "unprimed". A primed variable refers to its value at the *next* state, or its current-state value if it is unprimed. The use of primed variables in expressions allows for more expressive descriptions of state changes, especially when combined with the use of synchronous events (Sec. 2.3).

2.2 Operational Semantics Given a TTM module instance $\mathscr{M}$, an LTS (Labelled Transition System) is a 4-tuple $\mathscr{L} = (\Pi, \pi_0, \mathbf{T}, \rightarrow)$ where 1) Π is a set of system configurations; 2) $\pi_0 \in \Pi$ is an initial configuration; 3) $\mathbf{T}$ is a set of transitions names (defined below); and 4) $\rightarrow \subseteq \Pi \times \mathbf{T} \times \Pi$ is a transition relation.

We define E_{id} as the set of event transition names, and E_{fair} as the set of transition name prefixes, excluding values of demonic indices (i.e., including values of fair indices): $E_{id} \triangleq \{e, m \mid e \in E \wedge m \in e.f\_ind \rightarrow \text{VALUE} \bullet (e.id, m)\}$. On the one hand, we use $e(x)$ to denote the (external) transition name of event e with x, the values of its fair indices. On the other hand, when referring to the occurrence of e, in an LTL formula for instance, we use $e(x,y)$ to include y, the values of its demonic indices; otherwise, values of demonic indices are treated as internal non-deterministic choice within the event.

A configuration $\pi \in \Pi$ is defined by a 6-tuple (s, t, m, c, x, p), where:

• $s \in$ STATE is a value assignment for all the variables of the system. The state can be read and changed by any transition corresponding to an event in E.

• $t \in$ TIMER is a timer valuation function. Event transitions may start, stop, and read timers. A *tick* transition representing a global clock changes the timers.

• $m \in T \rightarrow$ BOOL records the status of monotonicity of each timer. Suppose event e_1 starts t_1, then we may specify that a predicate p becomes true within 4 ticks after e_1's occurrence. However, other events might stop or restart t_1 before p is satisfied, making t_1 not in sync with the global clock. The expression $m(t_1)$ (monotonicity of timer t_1) holds in any state where t_1 is not stopped or reset.

• $c \in E_{id} \to \mathbb{N} \cup \{-1\}$ is a value assignment for a clock implicitly associated with each event. These clocks are used to decide whether an event has been enabled for long enough ($c(e.id,x) \geq e.l$) and whether it is urgent ($c(e.id,x) = e.u$).

• $x \in E_{id} \cup \{\bot\}$ provides a sequencing mechanism: each transition e is immediately preceded by a transition $e\#$ to update the monotonicity record m.

• $p \in E_{id} \cup \{tick, \bot\}$ holds the name of the last event to be taken at each configuration. It is $\bot$ in the initial configuration. It allows us to refer to events in LTL formula, to state that they have just occurred.

We focus on components s and c that are affected the most by fair and demonic indices, whereas components t, m, and x, as to how the monotonicity status of timers is maintained, are less relevant and included in [10, Sec. 6].

Given a flattened module instance $\mathscr{M}$, transitions of its corresponding LTS are given as $\mathbf{T} = E_{id} \cup E\# \cup \{tick\}$, where $E\# \triangleq \{e \in E_{id} \bullet e\#\}$ is the set of monotonicity-breaking transitions as mentioned above. Explicit timers and event (lower and upper) time bounds are described with respect to this tick transition. We define the enabling condition of event $e \in E$ with fair index x and demonic index y as when its guard is satisfied, and when its implicit clock is in-between its specified bounds: ($e.en(x) \triangleq (\exists y \bullet e.grd(x,y)) \wedge e.l \leq c(e.id,x) \leq e.u$).

The initial configuration is defined as $\pi_0 = (s_0, t_0, m_0, c_0, \bot, \bot)$, where s_0 and t_0 come from the abstract (Sec. 2.1). The value of each event e_i's implicit clock depends on its guard being satisfied initially. More precisely, $c_0(e_i.id,x)$ equals 0 (the clock starts) if $(s_0,t_0) \models (\exists y \bullet e_i.grd(x,y))$[1]; otherwise, it equals -1.

An execution σ of the LTS L is an infinite sequence $\pi_0 \xrightarrow{\tau_1} \pi_1 \xrightarrow{\tau_2} \pi_2 \to \cdots$, alternating between configurations $\pi_i \in \Pi$ and transitions $\tau_i \in \mathbf{T}$. Below, we provide constraints on each one-step relation ($\pi \xrightarrow{e} \pi'$) in an execution. If an execution σ satisfies all these constraints then we call σ a *legal* execution. To characterize the complete behaviour of $\mathscr{L}$, we let $\Sigma_{\mathscr{L}}$ denote the set of all its legal executions. Given a temporal logic property φ and an LTS $\mathscr{L}$, we write $\mathscr{L} \models \varphi$ iff $\forall \sigma \in \Sigma_{\mathscr{L}} \bullet \sigma \models \varphi$. There are two possible transition steps (event $e(x)$ and $tick$):

$$(s,t,m,c,e(x),p) \xrightarrow{e(x)} (s',t',m',c',\bot,e(x)) \tag{1}$$

$$(s,t,m,c,\bot,p) \xrightarrow{tick} (s,t',m',c',\bot,tick) \tag{2}$$

Taking *e* The transition $e(x)$ specified in Eq. 1 is taken only if the x-component of the configuration is e (meaning that $e\#$ was just taken, so e is the only event allowed to be taken) and $(s,t,c) \models e.en(x)$. The component s' of the *next* configuration in an execution is determined non-deterministically by $e.action(x,y)$, which is a relation as demonic indices or assignments may be used. Consequently, any next configuration that satisfies the relation can be part of a valid execution, i.e., s' is only constrained by $(s,t,s') \in e.action(x,y)$. The following function tables specify the updates to c upon occurrence of transition $e(x)$.

For each event $e_i \in E$, $x \in e_i.f\_ind \to$ VALUE		$c'(e_i.id)$
$(s',t') \not\models (\exists y \bullet e_i.grd(x,y))$		-1
$(s',t') \models (\exists y \bullet e_i.grd(x,y))$	$(s,t) \models (\exists y \bullet e_i.grd(x,y)) \wedge \neg e_i = e$	$c(e_i.id,x)$
	$(s,t) \not\models (\exists y \bullet e_i.grd(x,y)) \vee e_i = e$	0

[1] If a state-formula q holds in a configuration π, then we write $\pi \models q$. For formulas such as guards which do not depend on all components of a configuration, we drop some of its components on the left of $\models$, as in $(s_0,t_0) \models e.grd(x,y)$.

We start and stop the implicit clock of e_i as a consequence of executing e, according to whether $e_i.grd$ just becomes or remains false (1st row), remains true (2nd row), or just becomes true (3rd row). Event e_i is ready to be taken if it becomes enabled $e_i.l$ units after its guard becomes true.

Taking *tick* The tick transition specified in Eq. 2 is taken only if the x-component of the configuration is $\perp$ (thus preventing *tick* from intervening between any $e\#$ and e pair) and if $\forall e \in E \bullet c(e.id,x) < e.u$.

For each event $e \in E$, $x \in e.f\_ind \rightarrow$ VALUE		$c'(e.id,x)$
$(s',t') \not\models (\exists y \bullet e.grd(x,y))$		-1
$(s',t') \models (\exists y \bullet e.grd(x,y))$	$(s,t) \not\models (\exists y \bullet e.grd(x,y))$	0
	$(s,t) \models (\exists y \bullet e.grd(x,y))$	$c(e.id,x)+1$

Thus, *tick* increments timers and implicit clocks towards their upper bounds.

Scheduling So far, we have constrained executions so that the state changes in controlled ways. However, to ensure that a given execution does not stop making progress, we need to assume fairness. The current TTM tool supports four possible scheduling assumptions.

1. Spontaneous event. When no fairness keyword is given, and the UTB is given as * or unspecified, then even when the event is enabled, it might never be taken.

2. Just event scheduling (a.k.a. weak fairness [9]). This is assumed when the event is declared with the keyword `just` and when the upper time bound is * or unspecified. For any execution $\sigma \in \Sigma_{\mathscr{L}}$, if an event e eventually becomes continuously enabled, then it occurs infinitely many times: $\sigma \models (\forall x \bullet \Diamond\Box e.en(x) \rightarrow \Box\Diamond(\exists y \bullet e(x,y)))$, where x ranges over e's fair indices and y its demonic indices.

This highlights the key distinction between fair and demonic indices. The fairness assumption guarantees that $e(x,\_)$ is treated fairly for every single value of x. For example, if x is a process identifier, making it a fair index means that as long as it is active, each process is eventually given CPU time. In contrast, if x is treated as a demonic index, then it is possible that infinitely often the same process will be given CPU time.

3. Compassionate event scheduling (a.k.a. strong fairness [9]). This is assumed when the event is declared with the keyword `compassionate` and when the upper time bound is * or unspecified. For any execution $\sigma \in \Sigma_{\mathscr{L}}$, if an event e becomes enabled infinitely many times, it has to occur infinitely many times. More precisely: $\sigma \models (\forall x \bullet \Box\Diamond e.en(x) \rightarrow \Box\Diamond(\exists y \bullet e(x,y))$.

4. Real-time event scheduling. The finite UTB $e.u$ of the event e is taken as a deadline: it has to occur within u units of time after $e.grd$ becomes true or after the last occurrence of e. To achieve this effect, the event e is treated as `just`. Since *tick* will not occur as long as e is urgent (i.e., $e.c = e.u$), transition e will be forced to occur (unless some other event occurs and disables it).

2.3 Semantics of Module Composition

So far we have specified the semantics of individual module instances. However, the TTM notation includes a composition. The semantics of systems comprising many instances is defined through flattening, i.e. by providing a single instance which, by definition, has the same semantics as the whole system.

Instantiation When integrating modules in a system, they first have to be instantiated, meaning that the module interface variables must be linked to global variables of the system which it will be a part of. For example if we have a *Phil* module (for philosopher) with two shared variables, *left_fork* and *right_fork*, and two global fork variables $f1$ and $f2$, we may instantiate them as:

instances *p1* = *Phil*(**share** *f1*, **share** *f2*) ; *p2* = *Phil*(**share** *f2*, **share** *f1*) **end**

Philosopher $p1$ is therefore equivalent to the module *Phil* with its references to *left_fork* substituted by $f1$ and its references to *right_fork* substituted by $f2$.

Composition The composition $m1||m2$ is an associative and commutative function on two module instances. To flatten the composition, we rename the local variables and events (by prepending the module instance name) so that they are system-wide unique. We then proceed to create the composite instance. Its local variables are the (disjoint) union of the local variables of the two instances. Its interface variables are the (possibly non-disjoint) union of the interface variables of both instances with their mode (`in`, `out`, `share`) adjusted properly [10, Table 1, p. 38] (e.g., variable `in x` in $m1$ and variable `out x` in $m2$ result in an `out` variable in the composite instance).

The simplest case of composition results in the union of the set of events of both instances. However, events from separate instances can be executed synchronously. This can be specified using the notation of synchronous events. As an illustration, consider a case where the plant and controller act synchronously.

```
module PLANT
  interface
    x : out INT = 0
  events
    generate
    do
      x :: 0 .. 10
    end
end
```

```
module CONTROLLER
  depends p : PLANT
  interface
    x : in INT
    b : out BOOL = false
  events
    respond sync p.generate as act
    do
      if x' > 0 then b := true else b = false fi
    end
end
```

```
instances
  env = PLANT(out x)
  c = CONTROLLER(in x, out b)
        with
          p := env
        end
  sync_env_c ::= env || c
end
composition
  system = sync_env_c
end
```

We say module *CONTROLLER* depends on module *PLANT*. At the module level (e.g., *CTRL*), we use a *depends* clause to specify a list of instances that the current module depends on. At the event level (e.g., *respond*), we use a *sync* ... *as* ... clause to specify the list of events to be synchronized, qualified by names of the dependent instances (e.g., *p.generate*), and to rename the synchronized events with a new name (*act*). Actions of events that are involved in synchronization may reference the primed version of input variables to obtain their next-state values. For example, the *respond* event uses the next-state value of the input variable x (i.e., *x'*) to compute the next-state value of its output variable b. In creating an instance, we use a *with* ... *end* clause to bind all its dependent instances, if any. We use the *::=* operator to rename the synchronized instances (e.g., *sync_env_c*). As instances *env* and *c* are synchronized as the new instance *sync_env_c*, taking the event *sync_env_c.act* has the effect of updating, as one atomic step, the monitored variable x then controlled variable b.

Specifying *depends* clauses (at the module level) and *sync* clauses (at the event level) results in one or more compound events whose actions are composed of those involved in the synchronization. We discuss the process of merging event actions below. For how event time bounds and fairness assumptions are merged in synchronization, refer to [10, p. 40].

The use of synchronous events results in three kinds of dependency graphs[2].

1. The *Module Dependency Graph* contains the set of vertices $V = MOD$, and the set of edges consisting of (m_1, m_2), where module m_1 depends on m_2.

In each connected component of the module dependency graph, we construct a *synchronous event set* (e.g., $\{PLANT.generate, CONTROLLER.respond\}$) by including each event e, where e declares a *sync* clause, and all events under e's *sync* clause.

[2]Assume that *MOD* denotes the set of declared modules, *EVT* the set of declared events qualified by their containing modules, e.g., *PLANT.generate*, and *VAR* the set of interface and local variables

2. An *Event Dependency Graph* contains the set of vertices $V = EVT$, and the set of edges consisting of (e_1, e_2), where e_1 and e_2 are in the same synchronous event set and e_2 is declared under the *sync* clause of e_1.

3. An *Action Graph* is constructed from each synchronous event set. We write VAR_s to denote variables that are involved in actions of events in a synchronous event set s. For each synchronous event set s, its corresponding action graph contains the set of vertices $V = VAR_s$, and the set of edges consisting of (v_1, v_2), where the computation of v_1's new (or next state) value depends on that of v_2. There are two cases to consider: 1) in an equation where v_2 appears on the RHS and v_1' on the LHS (i.e., $v_1' = \ldots\ v_2\ \ldots$); and 2) in an assignment where v_2 appears on the RHS and v_1 on the LHS (i.e., $v_1 := \ldots\ v_2\ \ldots$).

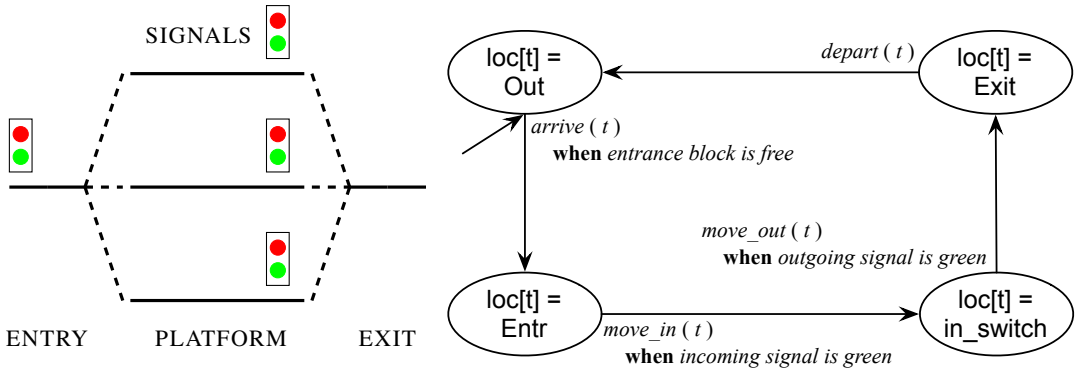
We perform a topological sort on each action graph to calculate the order of variable assignments, from which we calculate a sequence of variable projections. The *projection* for each variable v is a pair (v, act), where act is either an unconditional assignment (i.e., $v := exp$), or an conditional assignment (i.e., ***if*** b_1 ***then*** $v := exp_1$ ***elseif*** b_2 ***then*** $v := exp_2\ \ldots$ ***else*** $\ldots$). The latter case is resulted from the fact that changes on v (either through assignments or the primed notation) occur inside nested if-statements. Finally, the produced sequence of variable projections is adopted as the action of the compound event.

To ensure consistency, the TTM tool reports an error when, e.g., one of the above graphs contains a cycle, or a flattened (or compound) event assigns multiples values to the same variable.

Iterated Composition. Iterated composition allows us to compose an indexed set of similar instances. For example, in the case of a network of processes, we may specify the common process behaviour as a module once, and instantiate them from the set *PID* of process identifiers: $system = ||\ pid : PID\ @\ Process(\textbf{in}\ pid)$.

3 Example: A Train Control System

We illustrate the use of TTM indexed events in a train control system. There are two reasons for using the indexed events. First, all trains entering and leaving the station share a common behaviour. Second, by declaring event indices (ranging over trains) as fair, we can assert that individual trains arriving at the station are guaranteed to depart, without being blocked indefinitely by other trains.



(a) Topology

(b) State Transitions of Train $t \in TRAIN$

Figure 1: A Train Control System

Fig. 1a shows the topology of the train control system [3]. There is an entry block (*Entr*) and an exit block (*Exit*) on both ends of the station. Between the entry and exit blocks is a set *PLF* of special blocks

called platforms. At most one train may stay at the entry or exit block at a time. On the entry bock, there is a signal *isgn* regulating the incoming train, depending on the availability of platforms. On each platform $p \in PLF$, there is a signal $osgn[p]$ regulating the outgoing train, depending on the availability of the exit block. Fig. 1b illustrate the common behaviour of all trains. Each train is initially travelling outside the station. The train may first arrive at the entry block, provided that it is not occupied. When the signal *isgn* turns green, the train is directed via an in-switch to move in an available platform. For some train t, after it moved to platform p, it waits for the light signal of platform p to turn green and then moves away from p and onto the exit block. Then the train may depart from the station.

Trains must never collide in the train station. Also, once a train arrives, it should be eventually scheduled to depart from the station.

$$(\forall t1, t2 : TRAIN \bullet (t1 \neq t2 \wedge loc[t1] \neq Out \wedge loc[t2] \neq Out) \Rightarrow loc[t1] \neq loc[t2]) \quad (3)$$

$$\Box(loc[t] = Entr \Rightarrow \Diamond(loc[t] = Out)) \quad (4)$$

We consider two versions of TTM that satisfy both Eq. 3 and 4. Fig. 2a presents the TTM interface of an abstract version, where monitored and controlled variables are separated. As a result, the abstract version contains a single *STATION* module that: (a) owns all variables; and (b) mixes all events of train movement (e.g., event *move_out* in Fig. 3a) and of signal control (e.g., event *ctrl_platform_signal* in Fig. 4a). On the other hand, Fig. 2b presents the interface of a refined version, which distinguishes between one monitored variable (i.e., *occ* for the set of occupied platforms) and three controlled variables (i.e, *isgn* for an incoming train, *in_switch* for platform currently connected to the entrance block, and *osgn* for outgoing trains). Consistently, the behaviour of the controller and that of the trains are factored in separate events and placed in separate modules. The monitored variable (with modifier ***in***) is owned by the *STATION* module and read-only for the *CONTROLLER* module.

module *STATION*
interface
loc : **out ARRA** [*OPT_BLOCK*]
isgn : **out BOOL**
osgn : **out ARRA** [**BOOL**]
in_switch : **out** *BLOCK*

(a) Abstract

module *STATION*
interface
occ : **out ARRA** [**BOOL**]
isgn : **in BOOL**
osgn : **in ARRA** [**BOOL**]
in_switch : **in** *BLOCK*
local
loc : **ARRA** [*OPT_BLOCK*]

share initialization
qe : <*Queue*>
end
module *CONTROLLER*
interface
occ : **in ARRA** [**BOOL**]
isgn : **out BOOL**
osgn : **out ARRA** [**BOOL**]
in_switch : **out** *BLOCK*

(b) Refined: Separate Station & Controller Events

Figure 2: Train Control System in TTM: Interfaces

The refined version of TTM changes the representation of the data used by control events. In the abstract version (Fig. 2a), the array variable *loc* is used to map each train to its current location, constrained by type $OPT\_BLOCK \triangleq \{Out\} \cup BLOCK$ where $BLOCK \triangleq \{Entr,\ Exit\} \cup PLF$. All train events (e.g., *move_out* in Fig. 3a) are indexed with the set of trains and update their location accordingly (e.g., $loc[t] := Exit$). All control events (e.g., *ctrl_platform_signal* in Fig. 4a) query the value of *loc* in their guards (e.g., we write *!(||t: TRAIN @ loc[t] == Exit)* to check that the exit block is not occupied). However, a more realistic station controller may monitor platforms in the station only, rather than all trains including those travelling elsewhere outside the station. Consequently, in the refined version (Fig. 2b), by refactoring *loc* as a local variable in the *STATION* module (the environment), we hide it from the *CONTROLLER*. The controller then only has access to the monitored variable *occ* (i.e., the set of occupied platforms) which

encodes a coarser grain of information than *loc* (i.e., locations of all trains). Using the new monitored variable *occ* simplifies guards of controller events (Fig. 4b). Moreover, train events in the environment (e.g., Fig. 3b) updates both the local variable *loc* and the output variable *occ*. This raises the question of whether the *CONTROLLER* module accesses the monitored variable *occ* in a way consistent with the corresponding events in the abstract model. Therefore, we assert that a block is occupied if and only if it corresponds to the location of some train.

```
move_out(t : fair TRAIN) just
 when call(is_platform,loc[t]) && osgn[loc[t]]
 do loc[t] := Exit, osgn[loc[t]] := false end
```

(a) Abstract Version

```
move_out(t : fair TRAIN)[2, *] just
 when call(is_platform,loc[t]) && osgn[loc[t]]
 do loc[t] := Exit, occ[loc[t]] := false, occ[Exit] := true end
```

(b) Refined Version

Figure 3: Train Control System in TTM: the *move_out* Event in Module *STATION*

The two versions of TTMs are different in scheduling the green signals that control the passage from the platforms to the exit block. While the abstract model is non-deterministic about the order in which trains gain access to the exit block, the concrete model specifies the order uniquely. The signals are controlled by event *ctrl_platform_signal*. In the abstract version (Fig. 4a), the event is indexed by the set of trains. When the exit block is not occupied, more than one train located at a platforms may be eligible to move on to the exit block. To satisfy Property 4, we declare the index on trains as fair and adopt a strong fairness assumption (i.e., *compassionate*) on the controller event. That is, a train infinitely often qualified to leave the station does so eventually. However, such fairness assumption cannot be implemented efficiently. Consequently, in the refined version (Fig. 4b), we use a C# FIFO *Queue*[3] to specify the order of train departure. The reduced non-determinism allows us to remove the fair index on trains and weaken the fairness assumption (i.e., the event becomes *just*).

```
ctrl_platform_signal(p : fair BLOCK) compassionate
 when call(is_platform, p)
   && (&&p : BLOCK @ call(is_platform, p) -> !osgn[p])
   && !(||t: TRAIN @ loc[t] == Exit)
   && (||t: TRAIN @ loc[t] == p)
 do osgn[p] := true end
```

(a) Abstract Version in module *STATION*

```
ctrl_platform_signal just
 when qe.Count() != 0
   && !osgn[qe.First()]
   && !occ[Exit]
   && occ[qe.First()]
 do osgn[qe.First()] := true end
```

(b) Refined Version in module *CONTROLLER*

Figure 4: Train Control System in TTM: Controller Events

4 Example: Tabular Requirement of a Nuclear Shutdown System

We illustrate the use of synchronous events on parts of the software requirements of a shutdown system for the Darlington Nuclear Generating Station. We present two versions of the system. The first version presents a high-level requirements [11] where the controller responds instantaneously to environment changes. We synchronize the environment and controller events to model such instantaneity, and check

[3]Using a C# data object, implementation details of operations such as *Enqueue* are all encapsulated, resulting in a model simpler than one using a native TTM array.

it via an invariant property. The refined version illustrates how the response allowance [12] can be incorporated as event time bounds (i.e, the controller responds fast enough to environment changes). We decouple the controller from the environment, and check its response via a real-time liveness property.

Requirements of the shutdown system are described mathematically using tabular expressions (a.k.a. function tables) [4]. Figure 5 exemplifies tabular requirements for two units: Neutron OverPower (NOP) Parameter Trip (Figure 5a) and Sensor Trips (Figure 5b). In the first column, rows are Boolean conditions on monitored variables (i.e., input stimuli). In the second column, the first row names a controlled variable (i.e., output response); the remaining rows specify a value for that controlled variable. We use the formalism of tabular expressions to check the completeness (i.e., no missing cases from input conditions) and the disjointness (i.e., no input conditions satisfied simultaneously) of our requirements [4].

Condition	*Result* $c\_NOPparmtrip$
$\exists i \in 0..17 \bullet f\_NOPsentrip[i] = e\_Trip$	e_Trip
$\forall i \in 0..17 \bullet f\_NOPsentrip[i] = e\_NotTrip$	$e\_NotTrip$

(a) Function Table for NOP Controller

Condition	*Result* $f\_NOPsentrip[i]$
$calibrated\_nop\_signal[i] \geq f\_NOPsp$	e_Trip
$f\_NOPsp - k\_NOPhys < calibrated\_nop\_signal[i] < f\_NOPsp$	$(f\_NOPsentrip[i])_{-1}$
$calibrated\_nop\_signal[i] \leq f\_NOPsp - k\_NOPhys$	$e\_NotTrip$

(b) Function Table for NOP sensor i, $i \in 0..17$ (monitoring $calibrated\_nop\_signal[i]$)

Figure 5: Tabular Requirement for the Neutron Overpower (NOP) Trip Unit

The NOP Parameter Trip unit (the NOP controller) depends on 18 instances of the Sensor Trip units (the NOP sensors). There are two monitored variables for each NOP sensor i: (1) a floating-point calibrated NOP signal value $calibrated\_nop\_signal[i]$; and (2) a floating-point set point value f_NOPsp. The monitored signal is bounded by the two pre-set constants $k\_NOPLoLimit$ and $k\_NOPHILimit$. The monitored set point can be one of the four constants: $k\_NOPLPsp$ (low-power mode), $k\_NOPAbn2sp$ (abnormal mode 2), $k\_NOPAbn1sp$ (abnormal mode 1), and $k\_NOPnormsp$ (normal mode).

Each sensor i determines if the monitored signal goes above a safety range (i.e., $\geq f\_NOPsp$), in which case it trips by setting the function variable $f\_NOPsentrip[i]$ to e_Trip. To prevent the value of $f\_NOPsentrip$ from alternating too often due to signal oscillation, a hysteresis region (or dead band) with constant size k_NOPhys is created. The hysteresis region $(f\_NOPsp - k\_NOPhys, f\_NOPsp)$ is an open interval. When the monitored signal falls within this region, then the new value of $f\_NOPsentrip$ remains as that in the previous state, denoted as $f\_NOPsentrip_{-1}$. The NOP controller is responsible for setting the controlled variable $c\_NOPparmtrip$, based on values of $f\_NOPsentrip[i]$ from all its dependant sensors. If there is at least one sensor that trips, then the NOP parameter trips by setting $c\_NOPparmtrip$ to e_Trip.

According to the requirements, the system is initialized in a conservative manner. Each calibrated NOP signal is set to its low limit $k\_NOPLoLimit$, but each $f\_NOPsentrip[i]$ for sensor i and the controlled variable $c\_NOPparmtrip$ are all set to e_Trip. As we will see in our specification below (i.e., Equation 6), to ensure that the system satisfies the tabular specification in Figure 5, the NOP controller must have completed its very first response (denote as predicate $\neg init\_response$).

The requirements model in Figure 5 uses a finite state machine, with an arbitrarily small clock tick, that describes an idealized behaviour. At each time tick t, monitored and controlled variables are updated

instantaneously. State data such as $f\_NOPsentrip_{\_1}$ are stored and used for the next state. However, to make such requirements implementable, some allowance on the controller's response must be provided [12]. As a result, we present two versions of the NOP system in TTM: (1) an abstract version with plant and controller taking synchronized actions; and (2) a refined version with the response allowance incorporated as time bounds of the environment and controller events. The refined version allows us to assert timed response properties (e.g., once the monitored signal goes above the safety range, the controller trips within 2 ticks of the clock).

Abstraction of Input Signal Values. The TTM tool, like other model checking tools, cannot handle the real-valued monitored variables *f_NOPsp* and *calibrated_nop_signal*[*i*]. Instead, based on the given constants mentioned above, we partition the infinite domains of these two monitored variables into disjoint intervals. First, the four possible constant values for *f_NOPsp* have a fixed order and are bounded by constant low and high limits of the calibrated NOP signal. More precisely, we have 6 boundary cases to consider: $k\_NOPLoLimit < k\_NOPLPsp < k\_NOPAbn2sp < k\_NOPAbn1sp < k\_NOPnormsp < k\_NOPHiLimit$. Second, each of the four possible set points has an associated hysteresis band, whose lower boundary is calculated by subtracting the constant band size *k_NOPhys*, resulting in 4 additional boundaries[4] to consider: (a) $k\_NOPLPsp - k\_NOPhys$; (b) $k\_NOPAbn2sp - k\_NOPhys$; (c) $k\_NOPAbn1sp - k\_NOPhys$; and (d) $k\_NOPnormsp - k\_NOPhys$. Consequently, we have 10 boundary cases and 9 in-between cases (e.g., $k\_NOPLoLimit < signal < k\_NOPLPsp$) to consider. Accordingly, we construct a finite integer set *cal_nop* that covers all the 19 intervals.

For the purpose of modelling and verifying the NOP controller and sensors in TTM, we parameterize the system by a positive integer *N* denoting the number of dependant sensors.

Version 1: Synchronizing Plant and Controller. We first present an abstract version of the model that couples the NOP controller and its plant by executing their actions synchronously. Figure 6 illustrates the structure of synchronization. The dashed box in Figure 6 indicates the set of synchronized modules instances: plant *p*, controller *nop*, and 18 sensors *sensor_i* ($i \in 0\,..\,17$).

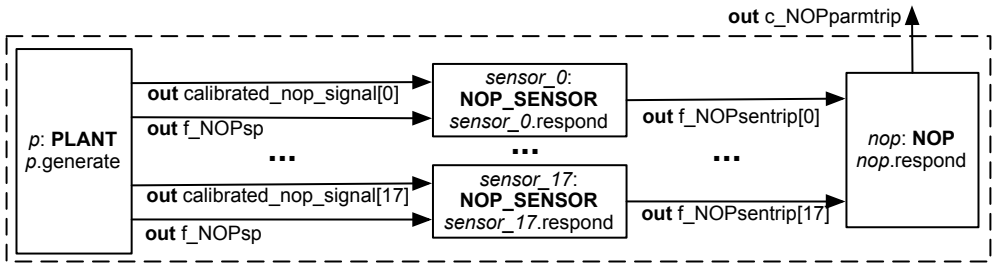


Figure 6: Neutron Overpower (NOP): Abstract Version – Synchronized Plant and Controller

Figure 8 (p. 95) presents the complete[5] TTM listing of the NOP unit as described above. The *generate* event of the plant non-deterministically updates the value of a global array that is shared with sensors attached to the NOP controller. The update is performed via the demonic assignment *calibrated_nop_signals :: **ARRAY***[*cal_nop*](*N*) (Lines 5 – 6). The NOP controller module (Lines 8 – 26) depends on two module instances (Lines 9–11). First, the controller depends on a plant *p* that generates

[4]Value of (a) is still greater than *k_NOPLoLimit*, and similarly value of (d) is still smaller than *k_NOPHiLimit*.

[5]For clarity, we present a version with one monitoring sensor. The full version with 18 sensors involves just declaring and instantiating additional dependent sensors. We also exclude definitions of constants and assertions.

an array of calibrated NOP signals (specified by the **out** array argument *calibrated_nop_signal* at Lines 4 and 47). Second, the controller depends on a sensor *sensor_0* that monitors a particular signal value (specified by the **in** argument *calibrated_nop_signal*[0] at Lines 30 and 48) and provides feedback (specified by the **share** argument *f_NOPsentrip*[0] at Line 31 and 48) for the central NOP controller to make a final decision (specified by the **out** argument *c_NOPparmtrip* at Lines 14 and 49).

Actions of the *respond* events of the NOP controller (Lines 19 – 24) and of its dependent sensors (Lines 36 – 43) correspond to the tabular requirements (Figure 5a and Figure 5b, respectively). We use primed variables in these actions to specify the intended flow of actions. Actions of the NOP sensor reference *f_NOP'* and *calibrated_nop_signal_i'* (Lines 37, 39, and 41) to indicate, that only after the instance *p* (in the same synchronous set) has written to these two variables can they be used to calculate the new value of *f_NOPsentrip*[*i*]. Similarly, actions of the NOP controller reference *f_NOPsentrip'*[*j*] (Lines 20 and 22) to indicate, that only after all sensor instances have written to this array can it be used to calculate the new value of *c_NOPparmtrip*.

We require that the *respond* event of the NOP controller, the *respond* events of its dependent sensors, and the *generate* event of the plant, are always executed synchronously (as a single transition). In declaring the controller event *respond*, we use a ***sync*** ... ***as*** ... clause to specify the events to be included in the synchronous set. When instantiating the NOP controller, we use a ***with*** ... ***end*** clause to bind its dependent plant and sensor instances (Line 49). Finally, we rename the synchronized plant, controller, and sensor instances for references in assertions (Line 50).

We check two invariant properties on this abstract version of NOP. First, as all dependent sensors have written to the shared array *f_NOPsentrip*, the NOP controller responds instantaneously.

$$\Box\left(\begin{array}{ll} & (\ \exists i : 0..N \bullet f\_NOPsentrip[i] = e\_Trip\ ) \Rightarrow c\_NOPparmtrip = e\_Trip \\ \wedge & (\ \forall i : 0..N \bullet f\_NOPsentrip[i] = e\_NotTrip\ ) \Rightarrow c\_NOPparmtrip = e\_NotTrip \end{array}\right) \tag{5}$$

Second, since all actions of the plant, the NOP controller, and sensors are synchronized together, we can assert that the controlled variable *c_NOPparmtrip* is updated as soon as the plant has updated the two monitored variables *f_NOPsp* and *f_NOPsentrip*.

$$\Box\left(\begin{array}{l} \left(\begin{array}{ll} & \neg\ init\_response \\ \wedge & f\_NOPsp = k\_NOPLPsp \\ \wedge & k\_NOPLPsp \leq calibrated\_nop\_signal[0] \leq k\_CalNOPHiLimit \end{array}\right) \\ \qquad \Rightarrow c\_NOPparmtrip = e\_Trip \end{array}\right) \tag{6}$$

However, the satisfaction of Equation 6 is an idealized behaviour without the realistic concern of some allowance on the controller's response [12]. That is, we shall instead allow the state predicate $c\_NOPparmtrip = e\_Trip$ to be established within a bounded delay.

Version 2: Separating Plant and Controller. We refine the TTM of NOP in Figure 8 by decoupling actions of the controller[6] and its plant. Figure 7 illustrates the refined structure of synchronization: the plant instance *p* is no longer synchronized with the controller. Consequently, the plant event *generate* and the synchronous controller event *respond* are interleaved.

The resulting system would fail to satisfy Equation 6, as we introduce some allowance on the response time (termed *response allowance* in [12]) of the NOP controller to environment changes. On

[6] In the NOP controller, actions of the NOP parameter trip unit and sensor units remain synchronized.

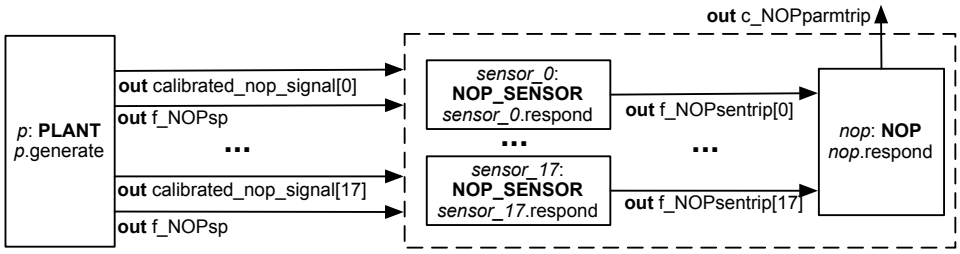


Figure 7: Neutron Overpower (NOP): Refined Version – Separate Plant and Controller

the other hand, as we still consider the controller's response actions, once initiated, take effect instantaneously, the resulting system should still satisfy Equation 5.

We apply the following changes to produce the refined TTM (Figure 8). First, in module *PLANT*, we revise time bounds of the *generate* event to $[2, *]$, which encodes the assumption that the controller (whose *respond* event has time bounds $[1, 1]$) responds fast enough to the environment changes. Second, in module *NOP*, we remove the declaration of *p : PLANT* as a dependent instance (Line 10). We also remove the declaration of *p.generate* as an event to be synchronized with the *respond* event (Line 17). Third, in creating the instance *nop* of module *NOP*, as it no longer depends on a *PLANT* instance, we remove the binding statement (Line 49), i.e., *env := env*. Fourth, in renaming the synchronous instance, we remove the plant instance (Line 50), i.e., *controller ::= sensor_0 || nop*. Finally, we add the plant instance into the composition (Line 52), i.e., *system = env || controller*.

By declaring a timer *t* and adding a ***start*** *t* clause to the *generate* event in module *PLANT* (Line 6), we can satisfy the following real-time response property:

$$\Box \left(\left(\begin{array}{ll} & f\_NOPsp = k\_NOPLPsp \\ \wedge & k\_NOPLPsp \leq calibrated\_nop\_signal[0] \leq k\_CalNOPHiLimit \\ \wedge & t = 0 \\ & \quad \Rightarrow mono(t) \ \mathbf{U} \ (\ c\_NOPparmtrip = e\_Trip \wedge t < 2 \ ) \end{array} \right) \right) \tag{7}$$

As soon as the set point value and monitored signal value are updated by the plant, the controller produces the proper response within two ticks of the clock. Before the controller responds, timer *t* must not be interrupted (i.e., reset by other events), so as not to provide an inaccurate estimate.

5 Discussion

Our new TTM notations facilitate the formal validation of cyber-physical system requirements. In the train control system (Sec. 3), the indexing construct allows us to select a specific actor (e.g., a train, a process, etc.) and specify a temporal property for that actor. Synchronous events, together with primed variables, allow us to check (real-time) response properties of the tabular requirements of a nuclear shutdown system (Sec. 4).

To our knowledge, the introduced notations of indexed events and synchronous events (and its combination with primed variables) are novel. For synchronous events, the conventional Communicating Sequential Processes (CSP) [7] and its tool [2] support multi-way synchronization by matching event names in parallel compositions. However, the conventional CSP does not allow processes to modify a

shared state. Instead, the system state can only be managed as parameters of recursive processes, making it impossible to synchronize events that denote different parts of simultaneous updates. The notations of un-timed CSP# and the stateful timed CSP (extended with real-time process operators such as time-out, deadline, etc.) [8] allow events to be attached with state updates. However, their semantics and tool support do not allow events that are attached with updates to be synchronized. The UPPAAL model checker and its language of timed automata [5] support the notion of broadcast channel for synchronizing multiple state-updating transitions (one sender and multiple receivers). However, the RHS of assignments can only reference values evaluated at the pre-state. There is no mechanism, such as the notion of primed variables supported in TTM, for specifying the intended data flow.

For indexed events, the verification tool support for both conventional CSP [7] and UPPAAL [5] does not allow for fairness assumptions. For UPAAL, it is likely to manually construct an observer, but this is likely to result in convoluted encoding in larger systems and thus is prone to errors. On the other hand, the PAT tool allows users to choose fairness assumptions at the event, process, or global level [9] for verifying the un-timed CSP# and stateful timed CSP [8]. However, our notion of indexed events are of finer-grained for imposing fairness assumptions, as we allow the declaration of event indices as fair.

References

[1] Patricia Derler, Edward A. Lee & Alberto Sangiovanni-Vincentelli (2012): *Modeling Cyber-Physical Systems*. *Proceedings of the IEEE (special issue on CPS)* 100(1), pp. 13 – 28. Available at http://dx.doi.org/10.1109/JPROC.2011.2160929.

[2] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov & AndrewW. Roscoe (2014): *FDR3 – A Modern Refinement Checker for CSP*. In: *TACAS*, *LNCS* 8413, Springer, pp. 187–201. Available at http://dx.doi.org/10.1007/978-3-642-54862-8_13.

[3] Simon Hudon & ThaiSon Hoang (2013): *Systems Design Guided by Progress Concerns*. In: *Integrated Formal Methods*, *LNCS* 7940, Springer, pp. 16–30. Available at http://dx.doi.org/10.1007/978-3-642-38613-8_2.

[4] Ryszard Janicki, DavidLorge Parnas & Jeffery Zucker (1997): *Tabular Representations in Relational Documents*. In: *Relational Methods in Computer Science*, Advances in Computing Sciences, Springer Vienna, pp. 184–196. Available at http://dx.doi.org/10.1007/978-3-7091-6510-2_12.

[5] Kim G. Larsen, Paul Pettersson & Wang Yi (1997): *UPPAAL in a Nutshell*. *International Journal on Software Tools for Technology Transfer* 1(1–2), pp. 134–152. Available at http://dx.doi.org/10.1007/s100090050010.

[6] JonathanS. Ostroff, Chen-Wei Wang, Simon Hudon, Yang Liu & Jun Sun (2014): *TTM/PAT: Specifying and Verifying Timed Transition Models*. In: *FTSCS*, *Communications in Computer and Information Science* 419, Springer, pp. 107–124. Available at http://dx.doi.org/10.1007/978-3-319-05416-2_8.

[7] A.W. Roscoe (2010): *Understanding Concurrent Systems*, 1st edition. Springer. Available at http://dx.doi.org/10.1007/978-1-84882-258-0.

[8] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi & Étienne André (2013): *Modeling and verifying hierarchical real-time systems using stateful timed CSP*. *ACM Trans. Softw. Eng. Methodol.* 22(1), pp. 3:1–3:29. Available at http://dx.doi.org/10.1145/2430536.2430537.

[9] Jun Sun, Yang Liu, Jin Song Dong & Jun Pang (2009): *PAT: Towards Flexible Verification under Fairness*. In: *CAV*, LNCS 5643, pp. 709 – 714. Available at http://dx.doi.org/10.1007/978-3-642-02658-4_59.

[10] C.-W. Wang, J. S. Ostroff & S. Hudon (2014): *Using Indexed and Synchronous Events to Model and Validate Cyber-Physical Systems*. Tech Report EECS-2014-03, York University.

[11] A. Wassyng & M. Lawford (2006): *Software tools for safety-critical software development*. *STTT* 8(4-5), pp. 337–354. Available at http://dx.doi.org/10.1007/s10009-005-0209-6.

[12] A. Wassyng, M. Lawford & X. Hu (2005): *Timing Tolerances in Safety-Critical Software*. In: *FM*, pp. 157–172. Available at http://dx.doi.org/10.1007/11526841_12.

```
1  module PLANT      /* Template for Nuclear Reactor */
2    interface
3      f_NOPsp : out INT = k_NOPLPsp
4      calibrated_nop_signal : out ARRA  [cal_nop](NUM_SENSORS) = [k_CalNOPLoLimit (NUM_SENSORS)]
5    events generate[1, 1]
6              do calibrated_nop_signal :: ARRA  [cal_nop](NUM_SENSORS), f_NOPsp := k_NOPLPsp end
7  end
8  module NOP      /* Template for Neutron Overpower Controller */
9    depends
10     env : PLANT
11     sensor_0 : NOP_SENSOR
12   interface
13     f_NOPsentrip : share ARRA  [y_trip](NUM_SENSORS)      /* shared, but read only */
14     c_NOPparmtrip : out y_trip = e_Trip
15   local after_init_response : BOOL = false
16   events
17     respond[1, 1] sync env.generate, sensor_0.respond as respond
18     do
19       after_init_response := true,
20       if (|| j: 0..(NUM_SENSORS−1) @ f_NOPsentrip'[j] == e_Trip) then
21         c_NOPparmtrip := e_Trip
22       elseif (&& k: 0..(NUM_SENSORS−1) @ f_NOPsentrip'[k] == e_NotTrip) then
23         c_NOPparmtrip := e_NotTrip
24       else skip fi
25     end
26 end
27 module NOP_SENSOR      /* Template for Sensors */
28   interface
29     f_NOPsp : in INT
30     calibrated_nop_signal_i : in cal_nop
31     f_NOPsentrip_i : share y_trip      /* shared, but write only */
32   local f_NOPsentrip_i_old : y_trip = e_Trip
33   events
34     respond[1, 1]
35     do
36       f_NOPsentrip_i_old' = f_NOPsentrip_i,
37       if f_NOPsp' <= calibrated_nop_signal_i' then
38         f_NOPsentrip_i := e_Trip
39       elseif (f_NOPsp' − k_NOPhys < calibrated_nop_signal_i') && (calibrated_nop_signal_i' < f_NOPsp') then
40         f_NOPsentrip_i := f_NOPsentrip_i_old
41       elseif calibrated_nop_signal_i' <= f_NOPsp' − k_NOPhys then
42         f_NOPsentrip_i := e_NotTrip
43       else skip fi
44     end
45 end
46 instances
47   env = PLANT (out f_NOPsp, out calibrated_nop_signal)
48   sensor_0 = NOP_SENSOR(in f_NOPsp, in calibrated_nop_signal[0], share f_NOPsentrip[0])
49   nop = NOP(share f_NOPsentrip, out c_NOPparmtrip) with env := env, sensor_0 := sensor_0 end
50   sys ::= env || sensor_0 || nop      /* named synchronous instance */
51 end
52 composition system = sys end
```

Figure 8: Requirement of NOP in TTM: Synchronized Plant and Controller