HAL
open science

Indefinite waitings in MIRELA systems

Johan Arcile,, Jean-Yves Didier, Hanna Klaudel, Raymond Devillers, Artur Rataj

To cite this version:

Johan Arcile,, Jean-Yves Didier, Hanna Klaudel, Raymond Devillers, Artur Rataj. Indefinite waitings in MIRELA systems. 4th International Workshop on Engineering Safety and Security Systems (ESSS 2015), Jun 2015, Oslo, Norway. pp.5–18, 10.4204/EPTCS.184.1 . hal-01230054

HAL Id: hal-01230054

https://hal.science/hal-01230054v1

Submitted on 12 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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]

To appear in EPTCS.

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 *Comp*→*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 $[min,max]$.

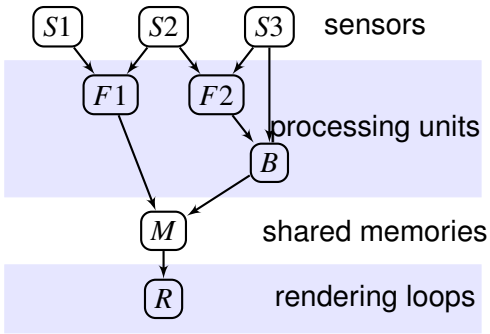
Example 1 *Let us consider the example corresponding to the* MIRELA *system specified in Figure 1 (top left), with three periodic sensors feeding two* First *processing units, and a* Both *one fed by the last sensor and the last* First*, and with a single rendering unit with its associated memory. The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

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



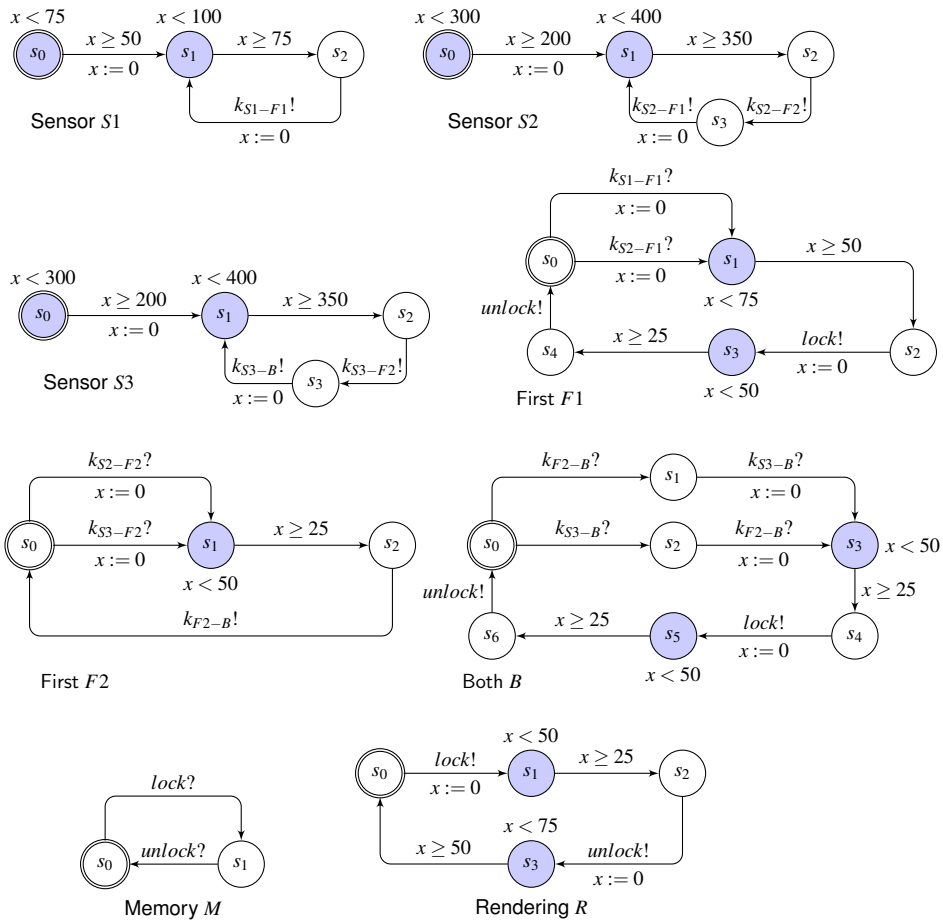


Figure 1: Specification, abstract scheme and TAST representation for Example 1. For Example 2, the TAST representation is as for Example 1 except for the invariant of location s_3 in Rendering R (which becomes $x < 100$) and the guard of its out-going arc (which becomes $x \geq 75$).

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.

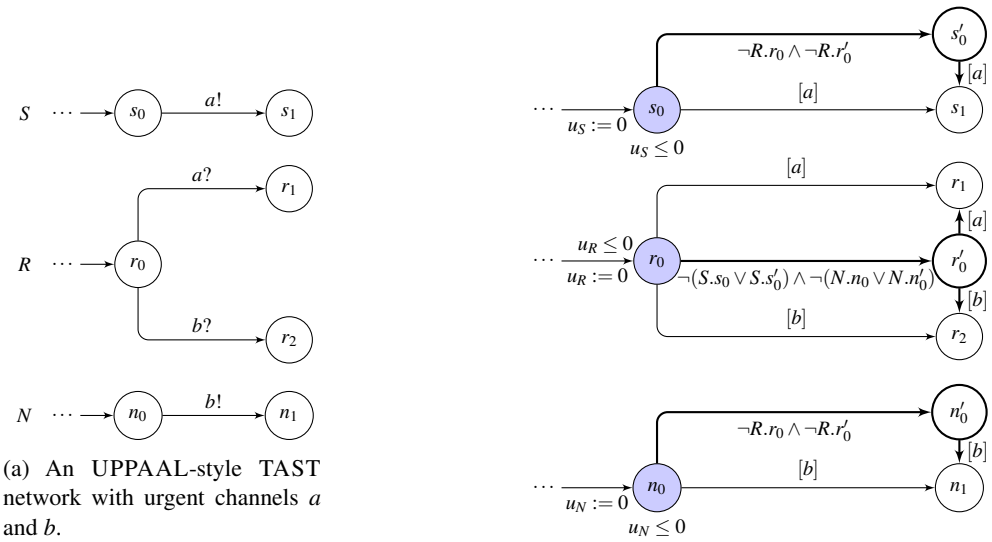
3 PRISM representation of MIRELA

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 $\neg(g^{k_1}_{loc} \vee \ldots \vee g^{k_m}_{loc})$, where $g^{k_i}_{loc} \equiv A_j.l_{k_i} \vee A_j.l'_{k_i}$ 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.

```
foreach w ∈ 𝒲 ∪ 𝒪 do
    Check φ_w ← EF EG w;
    if φ_w = false then
        w is neither a starvation, unbounded waiting nor deadlock;
    else
        if w ∈ 𝒪 then
            w is a starvation location
        else
            Check ψ_w ← EF AG w;
            if ψ_w = false then
                w is a starvation and/or an unbounded waiting
            else
                Check ρ_w ← EF EG (w ∧ (EF ¬w));
                if ρ_w = false then
                    w is a deadlock location
                else
                    w is a local deadlock, a starvation and/or an unbounded waiting location
                end
            end
        end
    end
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.