HAL
open science

Modelling and Analysing Mixed Reality Applications

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

To cite this version:

Johan Arcile, Tadeusz Czachórski, Raymond Devillers, Jean-Yves Didier, Hanna Klaudel, et al.. Modelling and Analysing Mixed Reality Applications. 4th International Conference on Man–Machine Interactions (ICMMI 2015), Oct 2015, Kocierz Pass, Poland. pp.3–17, 10.1007/978-3-319-23437-3_1 . hal-01230033

HAL Id: hal-01230033

https://hal.science/hal-01230033v1

Submitted on 14 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.

Modelling and Analysing Mixed Reality Applications

Johan Arcile[1], Tadeusz Czachórski[3], Raymond Devillers[2],
Jean-Yves Didier[1], Hanna Klaudel[1], Artur Rataj[3]

[1] Laboratoire IBISC, Université d'Evry-Val d'Essonne, France
johan.arcile@ens.univ-evry.fr,
{jean-yves.didier,hanna.klaudel}@ibisc.fr
[2] Département d'Informatique, Université Libre de Bruxelles, Belgium
rdevil@ulb.ac.be
[3] Institute of Theoretical and Applied Computer Science, Gliwice, Poland
tadek@iitis.pl, arturrataj@gmail.com

Abstract. Mixed reality systems overlay real data with virtual information in order to assist users in their current task. They generally combine several hardware components operating at different time scales, and software that has to cope with these timing constraints. MIRELA, for MIxed REality LAnguage, is a framework aimed at modelling, analysing and implementing systems composed of sensors, processing units, shared memories and rendering loops, communicating in a well-defined manner and submitted to timing constraints.
The framework is composed of (i) a language allowing a high level, and partially abstract, specification of a concurrent real–time system, (ii) the corresponding semantics, which defines the translation of the system to concrete networks of timed automata, (iii) a methodology for analysing various real-time properties, and (iv) an implementation strategy.
We present here a summary of several of our papers about this framework, as well as some recent extensions concerning probability and non–deterministic choices.

Keywords: mixed reality, timed automata, deadlocks, temporal properties

1 Introduction

The primary goal of a mixed reality (MR) system is to produce an environment where virtual and digital objects coexist and interact in real time. In order to get the global environment and its virtual or physical objects we need specific data, for which we shall use sensors (like cameras, microphones, haptic arms...). But gathering data is not sufficient as we want to see the result in our mixed environment; we then implement a rendering loop that will read the data and express the result in some way that a human can interpret (using senses like sight, hearing, touch). To communicate between those two types of components (sensors and renderers), shared memory units store the data, and processing units process the data received from sensors or processing units, and write them into shared memories or other processing units.

Since a few years, the MIRELA framework [8,7,14,9] (for MIxed REality LAnguage) is developed aiming at supporting the development process of applications made

of components which have to react within a fixed delay when some events occur inside or outside the considered area. This is the case in mixed reality applications which are evolving in an environment full of devices that compute and communicate with their surrounding context [6]. In such a context, it is difficult to keep control of the end-to-end latency and to minimise it. Classically, mixed reality software frameworks do not rely on formal methods in order to validate the behaviour of the developed applications. Some of them emphasise the use of formal descriptions of components inside applications in order to enforce a modular decomposition, possibly with tool chains to produce the final application [17,13], and ease future extensions [15] or substitutions of one module by another, like InTML [11,10]. Such frameworks do not deal with software failure issues related to time. On the contrary, this is the main focus of the MIRELA framework which proposes to use formal methods and automatic tools to analyse and understand potential issues, especially related to time, performance, and various kinds of bad behaviours such as deadlocks, starvations or unbounded waitings.

A typical modelling using MIRELA consists of the following stages. In the first phase, a formal specification of the system is given, in the form of a network of automata, defined using a high level description [8,9]. Then, depending on the applied approximation of the modelled system, and also on the properties we want to check, we decide how to transform the components. This may include theoretical issues, like the generation of bounding variant systems [7], which contain under– and over–approximating timed automata [1], and practical issues, like which model checker to target, depending on its capabilities. Finally, an implementation skeleton can be produced, e.g. in the form of a looping controller, which has a simple physical realisation, while certain properties of the original network are still met, which may be checked on to the chosen bounding variant systems.

2 MIRELA framework

Originally, the semantics of a MIRELA specification has been defined and implemented in UPPAAL [18] as a set of timed automata [1,2,3,19]. More precisely, we used a subclass called Timed Automata with Synchronised Tasks (TASTs) in order to cope with implementability issues (see [7] for details). TASTs are networks of timed automata, which communicate via urgent communication channels in the producer–consumer manner, and optionally contain wait locations, where the wait time $t \in [\min, \max]$ is non-deterministic. The communication dependencies between the automata form a directed and connected graph. There are also some additional constraints, so that the resulting automaton is non–Zeno, i.e. infinite histories taking no time or a finite time are excluded. For a complete description of TASTs see [9].

If the urgent channels are not available, like in the case of PRISM timed automata, a corresponding transformation is possible, which emulates the urgent channels [5].

Due to the verbosity of TASTs, MIRELA has its own, terse syntax, which can be automatically compiled into TASTs, but then also to representations adequate for other model checking tools than UPPAAL. The current MIRELA specification is presented in detail in [9]. Here we will give a summary.

$Ex1$:

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

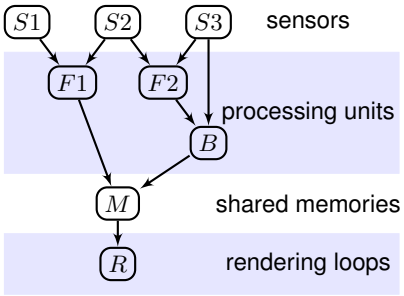


Fig. 1. Specification and information flow representation for our running example.

A network of components is defined as a list of declarations of the form:

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

Each declaration $Comp \rightarrow TList$ 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, Any of the components, after an optional initialisation, loops infinitely. Delays within the components use clocks, and clocks are never shared between components.

Here is a list of some of the standard components:

- two kinds of sensors that acquire data from outside and send it to processing units or memories:
  - $\mathsf{Periodic}(min_start, max_start)[min, max]$ starts with a one–off delay $\langle min_start, max_start \rangle$, then loops infinitely, each cycle lasting within $\langle min, max \rangle$;
  - $\mathsf{Aperiodic}(min_event)$ ascertains that the loop has a minimal delay of min_event, but no maximal delay is specified;
- three kinds of processing units that process data coming from possibly several different inputs (they can be combined in an acyclic hierarchy or form loops):
  - $\mathsf{First}(i_1[min,max], i_2[min,max], \ldots)$ which may have one or more inputs $i_1, i_2, \ldots$ and starts processing when data are received from one of them; the order is irrelevant; the loop delay depends on the input, as seen in the declaration, but a same interval may be distributed on many inputs;
  - $\mathsf{Both}(i_1, i_2)[min,max]$ which has exactly two inputs i_1, i_2 and starts processing when both input data are present; the loop delay is $\langle min, max \rangle$;
  - $\mathsf{Priority}(i_m[min,max], i_s[min,max])$, which has two inputs, master i_m and slave i_s, and starts processing when the master input is ready, possibly using the slave input if it is available before the master one; the delay specified at i_m is realised if the slave was not available; otherwise the delay at i_s is used.

– a shared memory Memory$(i_1[min,max], i_2[min,max], \ldots)$, with reads and writes locked by a common mutex, the write time depends on the input, as seen in the declaration;
– a rendering component Rendering$(min_rg, max_rg)(i_m[min,max])$ is a loop, which consists in reading a memory within a delay specified at i_m, then processes the read data within

This is illustrated with a running example, presented in Fig. 1 along with the corresponding flow of information. The corresponding TAST representation is depicted in Fig. 2.

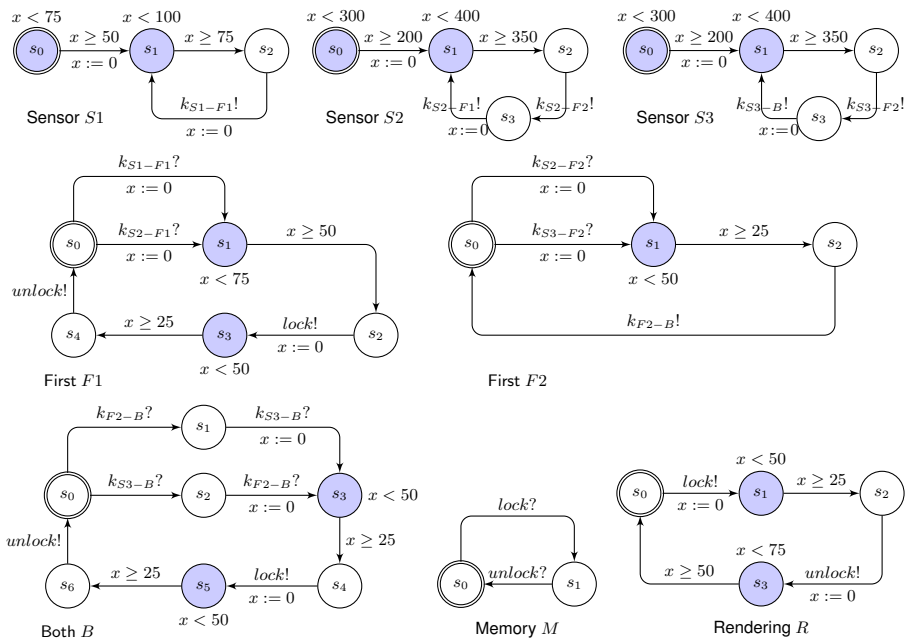


Fig. 2. TAST representation for our running example.

3 Analysis

We will discuss examples of proving various properties using Mirela, in particular related to bad behaviours, like various kinds of deadlocks and deadlock-like behaviours, which can be distinguished for timed automata.

A complete blocking occurs if a state is reached where nothing may happen: no location change is 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 available. A strong Zeno situation occurs when infinitely many location changes may be done without time passing. A weak Zeno situation occurs if infinitely many location changes may occur in a finite time delay. A weaker but potentially unpleasant situation occurs when a location change is available after some time, but this waiting time is unbounded. Those situations are usually considered highly harmful since an actually implemented system cannot meet these theoretical requirements.

For a network of timed automata (hence for TAST systems), a local deadlock occurs if, from some point, no location change is available for some component(s) while other components may evolve normally. A local unbounded waiting occurs if it is certain that a component will evolve, but the time before the component leaves its present state is unbounded. A starvation occurs if a component may be indefinitely blocked in some state (while not being deadlocked); this is different from the previous case, since here the time during which the component is blocked in the present state may really be infinite, and not simply unbounded. Notice however that starvations are not always to be avoided, for instance if the component corresponds to a failure handling.

3.1 Deadlock detection and graph analysis

While MIRELA has translation mechanisms allowing to use model checking tools, we may also take advantage of the specific features of the MIRELA systems, and in particular of the graph structure of the systems (see for example Fig. 1 – right).

First, one may observe that no (strong or weak) Zeno behaviour may happen since each loop in a component either contains a reset on some clock x and an arc with a guard $x \geq e$ (with $e > 0$), so that it is impossible to follow it in a null time, and it may be followed only a finite number of times in a finite delay, or contains arcs with input communications (in a memory cell for instance) and it may only progress indefinitely while communicating with a loop of the previous kind.

Moreover, if the system is well-formed, i.e., for each location with an invariant $x < e'$ each input arc resets x and each output arc has a guard $x \geq e$ with $e < e'$ (which is the case for MIRELA systems), it may not block the time.

A global deadlock (hence a complete blocking) may not occur in a complete system, i.e., having at least one memory unit and an associated rendering loop, since the rendering may indefinitely progress while accessing the memory (but starvation may occur if the memory is continually used by other units, and no fairness strategy is applied).

On the contrary local deadlocks may occur. They are intimately connected to the fact that processing units alternate two very distinct phases: first, some signals are received (reception phase), then some signals are emitted (emission phase) in a row (together with some synchronisations with memory units), and then the reception phase is resumed. Typically, in case of a cycle of processing units, it may happen that a processing unit in reception phase waits for a signal, which depends (directly or not) on its own emission, or symmetrically. There are also deadlocks of a mixed nature, combining components in emission and reception phases, that do not need involving any cycle of communicating units, contrary to what happened in the *emission* or *reception* cases. This is due to the fact that, when a component is blocked in its reception phase, the waiting condition corresponds to one or two arcs going in the reverse direction with respect

to the flow of information. Hence, a cycle of control may correspond (when there are both emitting and receiving blocked components) to a non cyclic flow of information.

In order to propose guidelines for the detection of local deadlocks, we may observe that a component may only deadlock in a wait location, but never when it waits for a $lock!$, $unlock!$ or $unlock?$. It also means that rendering loops never deadlock, and that memory units may not be the source of a deadlock: a memory unit may only deadlock if it has no rendering loop and all its users are deadlocked while trying to communicate with a non-memory unit.

Definition 1. *Let MS be a* MIRELA *system. An extended system $\overline{MS}$ is a temporal widening of MS if it has the same structure but each (or some) time interval $\mathcal{I}$ is replaced by another one $\overline{\mathcal{I}}$, where $\mathcal{I} \subseteq \overline{\mathcal{I}}$. Symmetrically, an extended system $\underline{MS}$ is a strengthening of MS if it has the same structure but each (or some) interval $\mathcal{I}$ is replaced by another one $\underline{\mathcal{I}}$, where $\underline{\mathcal{I}} \subseteq \mathcal{I}$.* □ 1

Note that, in particular, MS is a temporal widening and a strengthening of itself.

Proposition 1. *Let MS be a* MIRELA *system and $\overline{MS}$ be any of its temporal widenings. If a component does not deadlock at some location w in $\overline{MS}$, the same is true in MS. On the contrary, if $\underline{MS}$ is any strengthening of MS and a component deadlocks at some location w in $\underline{MS}$, the same is true in MS.* □

This result may be precious because the model checking complexity of timed systems depends of course on the complexity of the system, and on the formulas to be verified, but also on the $\gcd$ of the various constants occurring in the timing constraints, and on the various scales of the time intervals. Now, enlarging or restricting those constraints may considerably increase this $\gcd$, or uniformise the various time intervals. Note that, among the special enlargements that may be considered, some or all the upper bounds of time intervals may be replaced by ∞, and some or all the lower bounds of time intervals may be replaced by 1. In fact, it may be observed that lower bounds may even be replaced by 0: this may introduce Zeno behaviours, but does not kill existing deadlocks. Finally, one may observe that going from a constraint $x < e$ to $x \leq e$ is a form of a temporal widening, and going from $x < e$ to $x \leq e - 1$ is a form of strengthening.

Modifying the time intervals may be used to check the presence or absence of bad behaviours, but it may also be used to get a modified system, easier to model check, while maintaining its realistic aspect and its implementability.

Proposition 2. *Let MS be a* MIRELA *system. If a component deadlocks in MS, then so do all its input components, all its output* Both*, all its master output* Priority*, and all its output* First *units having a unique input. A memory component is never the source of a deadlock, but it may incur a deadlock propagation, if all its user components deadlock (which may not occur if there is a corresponding rendering loop).* □

Note that if a component deadlocks while it is a source of a slave input to a Priority, the latter does not necessarily deadlocks; a similar situation may occur if a First component has many inputs and one (or more, but not all) of them deadlocks, since it may

still manage inputs from non-deadlocking units. In both cases, the deadlock does not necessarily extend.

While temporal widening and strengthening do not modify the structure of a system, it is also possible to build modified (in general structurally simplified) systems. For instance, since rendering loops never deadlock, one may drop them. We may even also drop the memory units, since they are never the source of a deadlock.

Definition 2. *Let MS be a* MIRELA *system and C be one of its sensors or processing units. We shall denote by*

- *$\widehat{MS}$ the system obtained from MS by replacing all the time intervals by $[0, \infty)$, i.e., a form of untimed version of MS;*
- *$\widetilde{MS}$ the system obtained from $\widehat{MS}$ by dropping all its rendering loops;*
- *$C(\widetilde{MS})$ the part of $\widetilde{MS}$ (i.e., the set of components) which, in the abstract scheme, is connected to C without needing to traverse a memory unit; notice that it comprises the memory units.*
- *$C(\widetilde{\widetilde{MS}})$ the system $C(\widetilde{MS})$ without its memory units.* □ 2

Proposition 3. *Let MS be a* MIRELA *system and C a sensor or processing unit in it. Let $MS' = C(\widetilde{MS})$ and $MS'' = C(\widetilde{\widetilde{MS}})$. Then C deadlocks in $\widehat{MS}$ iff it deadlocks in MS' iff it deadlocks in MS''.* □

Combining Propositions 1 and 3 allows in some circumstances to detect the absence of local deadlocks while reducing the systems to be considered, both in terms of structure and temporal constraints. And if we add Proposition 2, this may even reduce the problem of detecting the absence of a local deadlock to the detection of the absence of global deadlocks in simplified systems, when the deadlock propagation ensures that a local deadlock extends to a global one. This may be useful since there are efficient algorithms to detect (the absence of) global deadlocks, and special commands to check it (in UPPAAL for instance).

Finally, we may observe that a local deadlock, in the original or temporally widened or strengthened model, may be handled by a model checker with query $\psi_w = \text{EF AG } w$, which checks if there is a situation (EF) where the considered component reaches w while there is no way (AG w) to get out of it: this thus corresponds to a local deadlock. Note that this formula pertains to CTL but uses a nesting of path formulas, so that it is not handled by UPPAAL and its optimised implementation. On the contrary, it is accepted by another model checker, PRISM, for which an automatic translation from MIRELA has also been developed (see Section 5). It is also possible to detect which components, at which locations, may be blocked simultaneously (recall that local deadlocks are due to many components blocking each others): if w_1 is a wait location in some component C_1, w_2 is in C_2, ..., one may use the same formula ψ_w, but where $w = w_1 \wedge w_2 \wedge \ldots \wedge w_k$, to check if $C_1, C_2, \ldots, C_k$ may be blocked simultaneously, in locations w_1, w_2, ..., w_k, respectively.

In summary, to detect the presence or absence of deadlocks, we may use a procedure, which first tries to get some information from simplified versions of the given system, using Propositions 3 and 2, and next uses Proposition 1 to try to get information on the remaining undecided wait locations.

3.2 Indefinite waiting detection

Even if there is no deadlock, it may happen that a component gets stuck in some location. In our case, there are essentially two sources of such an *indefinite waiting*.

For instance, this may occur with an aperiodic sensor, since no upper bound is specified for the time separating two successive data acquisitions. If we do not want that this propagates to other components, we should in particular avoid to use them in a Both unit, as a master to a Priority, or in a First when there is no other kinds of input. This may be qualified as an *unbounded waiting*, since the assumption is that the time between data acquisitions is finite, but unbounded.

Another kind of situation occurs if many components compete to communicate and, due to the non-deterministic way choices are performed, some of them never succeed. This then corresponds to a potentially infinite waiting, also termed *starvation*.

In a MIRELA system, this may for instance occur when performing a lock on a Memory unit (note that unlocks may only be performed by the components having succeeded in the lock): it may happen that a component (Rendering loop, Sensor or Processing unit) tries to access a Memory, fails because, when the memory is unlocked, the latter is attributed to another requesting component, and due to an unfortunate choice of the timing constraint (intervals), whenever the memory is unlocked, there are (remaining or new) requesting components and the considered one is never chosen, unfortunately, again and again. This also shows that, if we are concerned by starvations, in general it is not a good idea to consider systems simplified by removing Memory and Rendering units, since these can be essential ingredients for inducing infinite waitings.

This may be avoided by an adequate choice of the timing constraints, or by suitable fairness assumptions (and implementations) but in the latter case, ensuring that the waiting time will be finite does not necessarily imply that this time is (upper) bounded.

Proposition 4. *Let MS be a* MIRELA *system, a component may only incur an indefinite waiting in the initial activity location of an aperiodic sensor, or in a wait location, but never while waiting for an unlock.* □

Proposition 5. *Let MS be a deadlock-free* MIRELA *system, a component incurs an indefinite waiting at a location w iff the CTL formula $\phi_w =$* EF EG w *is true.* □

If w 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 wait locations, from which communications $k!$ or $k?$ only are offered (with $k \neq unlock$).

Since ϕ_w is a nesting of two path formulas, like ψ_w it may not be checked with UPPAAL. However, if we already know that w is reachable, instead of using this nested query, it is possible to use equivalently (up to contraposition) a *leads to* property, for which UPPAAL has an efficient algorithm: $\widetilde{\phi}_w = w{-}{-}{>}\neg w$ means that, if true, after w we shall eventually leave it, i.e., we shall have no deadlock and no indefinite waiting in w. Again, instead of working directly on the original system, one may consider temporally widened and/or strengthened versions of it.

Proposition 6. *Let MS be a* MIRELA *system, $\overline{MS}$ be one of its temporal widenings (while avoiding to start an interval from* 0*), and $\underline{MS}$ be one of its strengthenings. If a component incurs an indefinite waiting at some location w in MS, the same is true in $\underline{MS}$, and if a component incurs an indefinite waiting at some location w in $\underline{MS}$, the same is true in MS.* □

Note that, here, we may not drop memory units and/or rendering loops, since the latter may be needed ingredients to cause an indefinite waiting. A procedure to detect an indefinite waiting of a component C at location w may thus be proposed. If an indefinite waiting is found for a temporally widened system, nothing may be inferred in general on the original system; however, it may happen that in some circumstances a closer (manual) analysis of the found indefinite wait reveals that the same situation occurs in the original system: then the procedure may be stopped with a positive answer.

If the system presents deadlock situations (checkable with ψ), we could wonder if it also presents indefinite waitings. If ψ_w is false while ϕ_w is true, clearly we have an indefinite waiting at location w. But if ψ_w is true, it could still happen that the system also presents an indefinite waiting at the same location, but for a different environment than the deadlock. This may be checked by a slightly more elaborate CTL formula: $\rho_w = \text{EF EG } (w \wedge (\text{EF } \neg w))$.

Proposition 7. *Let MS be a* MIRELA *system. It presents an indefinite waiting at some location w iff ρ_w is true.* □

3.3 Starvation viz. unbounded waiting detection

If a system has no aperiodic sensor, it is sure that there is no unbounded waiting and that any found indefinite waiting (hence formula ρ_w) corresponds to a starvation phenomenon. The same is true if there are aperiodic sensors, but it is sure their unbounded delays do not propagate. But otherwise we could want to know if there are pure starvations and/or pure unbounded waitings, even for a same location (but for different environments).

Let us assume the considered specification MS presents n aperiodic sensors and let us denote by $a_1, a_2, \ldots, a_n$ their respective initial locations (in the TAST translation), and that there is a possible propagation of unbounded waitings.

To check a starvation in a wait location w, we may use the following property 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 (or another one), one may use the formula: $\zeta_w = \text{EF } ((\text{EG } w) \wedge \text{A}((\text{G } w) \Rightarrow (\text{FG } a_1) \vee \cdots \vee (\text{FG } a_n)))$ 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 waiting in w.

Unfortunately, those last two formulas belong to CTL* and, while it is known that CTL* is decidable, the corresponding decidability algorithm is extremely intricate, and we do not know any implementation of it.

4 Temporal properties

Another kind of question that may be asked on such systems concerns the minimal and/or maximal durations taken by components to perform their operations. We may be interested in exact values, or in bounds such as: "is this time greater than n units" or "is it between n and m". For instance, one may ask how much time a component may wait in some location until a *rendez-vous* is performed, one may wonder how much time a component takes to perform its (main) loop, or to go from one location to another one. We may need for that to add a new clock y that is reset when we enter the starting location, and then we check the greatest and smallest values of that clock when reaching the goal location s. Since the greatest value is calculated when we leave s, if we want to know that value when we enter it, a general trick to do it in UPPAAL is to add an *urgent location* s^u before s, i.e., a location where time may not progress (an urgent location is time-freezing), while redirecting the input arcs of s to s^u. The same trick will avoid to consider for the minimal value the initial value 0 when the goal location is the initial one. A typical query to do that with UPPAAL is `sup{C.`s^u`}:y` (resp. `inf{C.`s^u`}:y`) which determines the supremum (resp. infimum) of y when entering location s in component C.

However, in some complex cases such a method reveals inefficient due to the large number of states of the system, and an abstraction method like the following may be useful. The general idea is to consider iteratively some simplified, temporally widened systems, on which the duration bounds estimation is feasible, and to use the obtained bounds to get better temporal widenings, hence to progressively improve the bounds until either no improvement is possible or the obtained bounds are considered satisfactory. To get such simplified systems, the idea is to cut the original system into parts, with some components in the common boundaries, and no communication between components in the interior of different parts (the communications with the exterior must go through the boundaries). Then, one considers iteratively each part (let us call it $\mathcal{P}$) and we isolate it by replacing each connection between $\mathcal{P}$ and the other parts by an activity location associated with an interval encompassing (hence the temporal widening) the interval in which the actual connection takes place. By analysing the durations in the simplified system for $\mathcal{P}$, we shall get intervals encompassing the durations of the interactions between the other parts and $\mathcal{P}$. When we shall consider another part $\mathcal{P}'$, these intervals will be used for abstracting the interactions between $\mathcal{P}'$ and $\mathcal{P}$.

5 Support for PRISM

Besides the translation of MIRELA specifications into TASTs, in a form adequate to use the UPPAAL model checker, another translation mechanism has been devised to the input format of PRISM[12], another model checking framework, able to analyse probabilistic systems but also non-deterministic ones. In particular, the *digital clocks engine* of PRISM accepts CTL requests that UPPAAL doesn't. However, the translation is less easy than to TAST, due to various characteristics of the models:

- Discrete clocks: the digital clocks engine 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;

- 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. 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;
- 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. This was done by duplicating some locations and by introducing adequate guards [5].

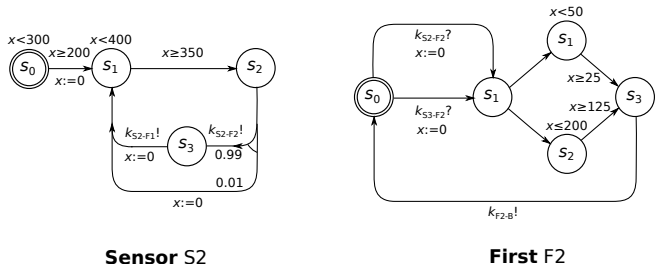
This was realised by adding a dedicated plugin to the translator J2TADD [16] aiming at producing specifications for the PRISM probabilistic model checker.

This may also be a gateway to add probabilities to the framework, hence to turn non-deterministic MIRELA models into stochastic ones. This may be done at two different levels. First, one may replace the time intervals $\min - \max$ by (continuous or discrete) probability distributions. Next, when many synchronisations are offered to a component at some point (which may happen with a non-null probability when discrete distributions are used), a random drawing mechanism must be defined to perform the choice. It is also possible to mix probabilistic and nondeterministic features, and to allow for alternative paths.

Let us discuss a possible example of such an extension, by modifying two components in Ex1, as illustrated in Fig.3(a). We would thus have a sensor S2 which skips data transmission at a given ratio. Let r be the skipping probability: this could be specified by an optional modifier `drop`=r after the list of output components of the sensor. We also have a processing unit F2, which undergoes, in an unspecified manner, occasional time–consuming clean–up computations. This could be specified by a $+$ operator between the various execution possibilities.

In effect, in the resulting system $Ex1'$, whose specification is shown in Fig. 3(b), the sensor S2 now skips every 100th data transmission on average, and the processing unit F2 occasionally, in an unspecified manner, undergoes an additional clean–up computations, which last from 100 to 150 ms.

An example of a probabilistic property, against which we could check $Ex1'$, using Prism's parametric model checking, and replacing the constant `drop`=0.01 with and undefined value `drop`=R, might be "how the value of R affects the maximum probability that a deadlock will occur in the first 10 time units". If the result would be a function with R absent or reducible, it would mean that the drop does not affect the probability of the deadlock.



(a)

$Ex1'$:

$$
\begin{aligned}
S1 &= \mathsf{Periodic}(50,75)[75,100];\\
S2 &= \mathsf{Periodic}(200,300)[350,400]{\rightarrow}(F2,F1)\ \text{drop} = 0.01;\\
S3 &= \mathsf{Periodic}(200,300)[350,400]{\rightarrow}(F2,B);\\
F1 &= \mathsf{First}(S1, S2[50,75]);\\
F2 &= \mathsf{First}(S2, S3[25,50] + [125,200]);\\
B &= \mathsf{Both}(S3, F2)[25,50];\\
M &= \mathsf{Memory}(F1[25,50], B[25,50]);\\
R &= \mathsf{Rendering}(50,75)(M[25,50]).
\end{aligned}
$$

(b)

Fig. 3. An example of Mirela components, which would replace the respective automata in Fig. 2, and the corresponding specification of the modified example.

6 Implementation strategy

Given a verified MIRELA specification MS, which satisfies some properties considered important, the idea is to use this specification for producing an implementation aiming at preserving those properties. The approach from [7] considers implementation prototypes, which take the form of a looping controller $\mathcal{C}_{MS,\Delta}$, obtained from the TAST representation of MS and parameterised with a well-chosen sampling period Δ. Such a controller may execute zero or several actions in the same period Δ. Obviously, there are semantic differences between the implementation and the specification, coming mostly from the interpretation of the continuous clock values in the sampled world of the implementation and the immediate reaction of the system when a synchronisation becomes possible. For example, one may easily observe that even if the original specification MS does not reach some error state, the controller $\mathcal{C}_{MS,\Delta}$ may reach it because the sampling allows to evaluate transition conditions potentially larger than it was the case in MS. The approach then proposes an over-approximating model of the implementation in order to check if the essential properties of the original specification are still satisfied by the implementation. The new model "covers" the evolutions of the controller $\mathcal{C}_{MS,\Delta}$, hence "sandwiching" the implementation between the original

specification and this auxiliary model. This model, $\overline{MS}$, is very similar to MS, but with relaxed timing constraints, and essentially allows to check if the safety properties of the specification are preserved by the implementation.

7 Conclusion and future work

The analysis techniques described above have been checked on various realistic examples, with satisfactory results [5,4]. For instance, when analysing our running example, the evaluation of $\tilde{\phi}_w$ with UPPAAL takes a few seconds for the various interesting locations w, while the evaluation of ψ_w and ρ_w with PRISM takes a hundred of seconds. Similarly, obtaining bounds for the looping time of each component took a few seconds, with UPPAAL and the abstraction technique.

In the future, we plan to extend the MIRELA specification by approximate probabilistic distributions, and also by explicit state variables.

Acknowledgement

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

References

1. R. Alur and D. L. Dill. Automata for modeling real-time systems. In *International Colloquium on Algorithms, Languages, and Programming (ICALP) 1990*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.
2. R. Alur and D. L. Dill. The theory of timed automata. In *Real Time: Theory in Practice (REX Workshop)*, volume 600 of *LNCS*, pages 45–73. Springer, 1991.
3. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
4. J. Arcile, J.-Y. Didier, H. Klaudel, R. Devillers, and A. Rataj. Analysis of real-time properties of mixed reality applications. *Submitted paper*, 2015.
5. J. Arcile, J.-Y. Didier, H. Klaudel, R. Devillers, and A. Rataj. Indefinite waitings in mirela systems. In *Engineering Safety and Security Systems (ESSS 2015), workshop of the 20th International Symposium on Formal Methods*, 2015.
6. M. Chouiten, C. Domingues, J.-Y. Didier, S. Otmane, and M. Mallem. Distributed mixed reality for remote underwater telerobotics exploration. In *Virtual Reality International Conference, VRIC '12*, pages 1:1–1:6, France, March 28-30 2012. ACM.
7. R. Devillers, J.-Y. Didier, and H. Klaudel. Implementing timed automata specifications: The "sandwich" approach. In *13th International Conference on Application of Concurrency to System Design (ACSD 2013)*, pages 226–235. IEEE, 2013.
8. J.-Y. Didier, B. Djafri, and H. Klaudel. The mirela framework: modeling and analyzing mixed reality applications using timed automata. *Journal of Virtual Reality and Broadcasting*, 6(1), 2009.
9. J.-Y. Didier, H. Klaudel, M. Moine, and R. Devillers. An improved approach to build safer mixed reality systems by analysing time constraints. In *Proceedings of the 5th Joint Virtual Reality Conference*, 2013.

10. P. Figueroa, W. F. Bischof, P. Boulanger, H. J. Hoover, and R. Taylor. Intml: A dataflow oriented development system for virtual reality applications. *Presence: Teleoperators and Virtual Environments*, 17(5):492–511, 2008.
11. P. Figueroa, J. Hoover, and P. Boulanger. Intml concepts. *University of Alberta. Computing Science Department, Tech. Rep*, 2004.
12. M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, 2004.
13. M. E. Latoschik. Designing transition networks for multimodal vr-interactions using a markup language. In *Proceedings of the 4th IEEE International Conference on Multimodal Interfaces*, page 411. IEEE Computer Society, 2002.
14. M. Moine. Implementation tool of timed automata specifications. Master's thesis, ENSIIE – Université d'Evry-val d'Essonne, 2013.
15. D. Navarre, P. Palanque, R. Bastide, A. Schyn, M. Winckler, L. P. Nedel, and C. M. Freitas. A formal description of multimodal interaction techniques for immersive virtual reality applications. In *Human-Computer Interaction-INTERACT 2005*, pages 170–183. Springer, 2005.
16. A. Rataj, B. Wozna, and A. Zbrzezny. A translator of java programs to tadds. *Fundam. Inform.*, 93(1-3):305–324, 2009.
17. C. Sandor, T. Reicher, et al. Cuiml: A language for the generation of multimodal human-computer interfaces. In *Proceedings of the European UIML conference*, volume 124, 2001.
18. Uppaal. http://www.uppaal.org/.
19. M. T. B. Waez, J. Dingel, and K. Rudie. Timed automata for the development of real-time systems. Research Report 2011-579, Queen's University – School of Computing, Canada, Aug. 2011.