Deadlock and temporal properties analysis in mixed reality applications

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

Deadlock and temporal properties analysis in mixed reality applications

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

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

***Abstract*—Mixed reality systems overlay real data with virtual information in order to assist users in their current task; they are used in many fields (surgery, maintenance, entertainment,...). Such systems 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 paper describes how harmful software behaviour, which may result in possible hardware deterioration or revert the system's primary goal from user assistance to user impediment, may be detected such as (global and local) deadlocks or starvation features. This also includes a study of temporal properties resulting in a finer understanding of the software timing behaviour, in order to fix it if needed.**

***Keywords*—Mixed reality; timed automata; deadlocks; temporal properties;**

I. 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 are used to 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. As usual, in order to keep the system consistent we observe a rule that will not allow to write and read concurrently a same memory cell. An example of the decomposition schema of an MR application, together with the flow of information, is shown in Figure 1.

Since a few years, the MIRELA framework (for MIxed REality LAnguage [8], [7], [16], [9]) 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]. Mixed reality set-ups are prone to various issues related to time that are needed to be solved since they use sensors to acquire knowledge from the real world, process acquired data and then display the results to the system's operator using rendering loops. All these devices are possibly running with different time constraints and often relying on multi-threading techniques in order to optimise the available computational power. 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 like those cited in [5], [19], [13], [18], [14], [10] 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 [20], [15], and ease future extensions [17] or substitutions of one module by another, like InTML [12], [11]. 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 time related issues.

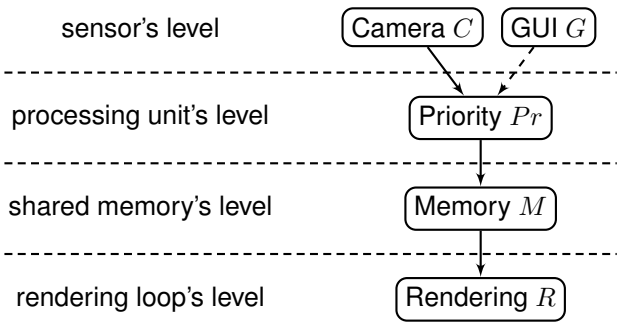


Fig. 1. Decomposition schema of an MR architecture (Example 1).

Nowadays, in order to cope with time constraints when developing MR applications, practitioners rely mostly on fast response and high performance hardware, even if this does not ascertain that critical constraints will always be respected. Also, being super-fast may be non-necessary and contradictory with other aspects, like cost or energy saving. Validating the application before testing it on the actual hardware, by modelling it and applying formal method techniques to prove its robustness, appears as a particularly interesting approach. Actually, it may avoid catastrophic situations and unnecessary hardware deterioration costs, by allowing to identify design errors in an early stage, and to correct them in due time.

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, and represents an ideal world, for which some properties considered important may be checked. The second phase concerns the analysis of the system: it essentially consists in model-checking a set of desired properties. It may be performed using existing verification tools like UPPAAL [21]. 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. Such a prototype is 'sandwiched' between the original specification and an auxiliary over-approximating model, on which one may verify if the expected properties are still satisfied.

In this paper, we revisit the first two phases of the MIRELA framework: in the first phase, we introduce a formal notation for MR time related specifications together with an automatic translation into a network of timed automata (Sections II–IV). In the second phase (Section V), we focus on the analysis of properties of such specifications such as deadlocks, starvation of components, and temporal properties such as minimum response time. In particular, we show that in some cases local and global deadlocks may be found or excluded by an essentially syntactic analysis of the specification. In the case of an existing deadlock, we provide a guideline for removing it while preserving as much as possible the desired intuitive behaviour (Section VI).

II. MIRELA SYNTAX AND EXAMPLES

A MIRELA specification is a list of components of the form:

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

where a component $Comp$ is either a sensor $Sensor$, a processing unit $PUnit$, a shared memory unit $MUnit$ or a rendering loop $RLoop$. A $TList$ 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; it is not requested 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].

Components are specified following the syntax:

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

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.

We consider two kinds of sensors (without source, with processing or memory units as targets):

- 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 (PU) process data coming from possibly several different sources of data. It is also possible to combine them (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 period between min_rg and max_rg time units, and the processing of data has a duration in the interval $[min,max]$.

This language has been carefully tailored in order to capture exactly the needed features of MR applications. To illustrate our methodology, we shall use the following simple but rather realistic examples.

Example 1: A classical setup for an MR application: The components are two sensors: a periodic sensor (camera C) and an aperiodic sensor (graphical user interface G), a shared memory M, a priority processing unit Pr and a rendering loop R. Its textual representation is given below while its component architecture is sketched in Figure 1.

[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.

Ex_1 :

$$\begin{array}{rcl} C & = & \mathsf{Periodic}(200,300)[350,450]; \\ G & = & \mathsf{Aperiodic}(20); \\ Pr & = & \mathsf{Priority}(C[250,350],G[250,350]); \\ M & = & \mathsf{Memory}(Pr[20,30]); \\ R & = & \mathsf{Rendering}(50,75)(M[21,31]). \end{array}$$

Example 2: A cascade of processing units:

Ex_2:

$$\begin{array}{rcl} C & = & \mathsf{Periodic}(2000,3000)[3500,4500]; \\ I & = & \mathsf{Periodic}(400,600)[900,1000]; \\ U & = & \mathsf{First}(C[2000,3000])\rightarrow(B,M); \\ B & = & \mathsf{Both}(I,U)[50,75]; \\ M & = & \mathsf{Memory}(B[200,300],U[300,400]); \\ G & = & \mathsf{Rendering}(500,750)(M[210,310]). \end{array}$$

The corresponding scheme is given in Fig. 2. Notice that processing unit First U sends data to B (which processes it and sends the result to M) and also directly to M:

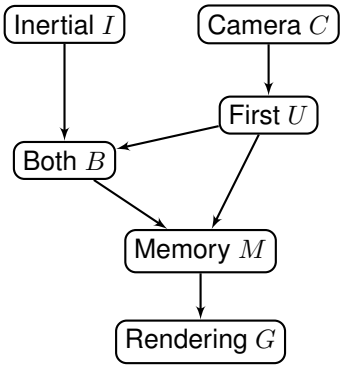


Fig. 2. Scheme of Example 2.

Example 3: An application containing a cycle of processing units:

Ex_3 :

$$\begin{array}{rcl} C & = & \mathsf{Periodic}(2000,3000)[3500,4500]; \\ I & = & \mathsf{Periodic}(400,600)[900,1000]; \\ U & = & \mathsf{First}(C[7000,9000])\rightarrow(M,L); \\ F & = & \mathsf{First}(L[40,60],I[40,60]); \\ L & = & \mathsf{First}(F[20,30],U[20,30])\rightarrow(M,F); \\ M & = & \mathsf{Memory}(U[300,400],L[10,20]); \\ G & = & \mathsf{Rendering}(500,750)(M[310,420]) \\ H & = & \mathsf{Rendering}(80,120)(M[20,30]). \end{array}$$

The corresponding scheme is represented in Fig. 3.

III. Timed Automata with Synchronised Tasks

Since their introduction in [1], [2], [3] timed automata have been widely used to model complex real time systems and to check their temporal properties. Since then many variants have been considered [22]. In order to get a compositional aspect, UPPAAL starts from a network of timed automata from which a synchronised product may be constructed to get a more classical timed automaton. We shall not need the full generality of the timed automata allowed by UPPAAL however, but a subset called Timed Automata with Synchronised Tasks (TASTs) in order to cope with implementability issues (see [7] for more details).

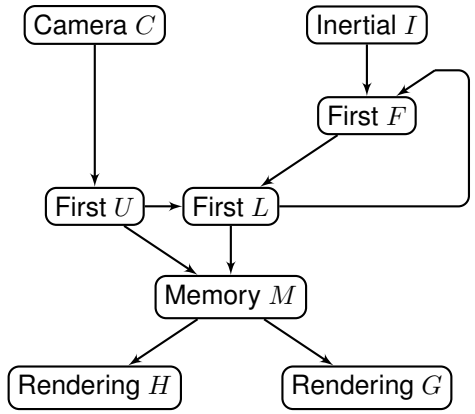


Fig. 3. Scheme of Example 3.

A. TAST's features

Syntactically, a timed automaton is an annotated directed (and connected) graph, with an initial node, provided with a finite set of non-negative real variables called *clocks*, initially 0, increasing with time and reset when needed; we shall not allow to share clocks between automata. The nodes (called *locations*) are annotated with *invariants* (predicates allowing to enter or stay in a location). Since we aim at describing systems of sensors and actuators, we shall distinguish the locations associated with an internal activity and the locations where one waits for some event or contextual condition. 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 of the form $x < e$ or empty (= true = $x < \infty$), with $0 < e' < e$. For a waiting location, all the output arcs have a communication action $k!$ (output) or $k?$ (input), allowing to glue together the various automata composing a system, since they must occur by input-output pairs. In order to structurally avoid Zeno evolutions (i.e., infinite histories taking no time or a finite time), we shall finally assume that each loop in the graph of the automaton presents (at least) a constraint $x \geq e$ in a guard (recall that e is strictly positive) and a reset of x for some clock x, or contains only input channels ($k?$).

In figures, locations will be represented by round nodes, the initial one having a double boundary, and activity locations are indicated by light blue background colour.

IV. TAST components for MIRELA

General TAST representations of MIRELA components are assembled in Figure 4. Synchronisations with memories are performed by $lock/unlock$ pairs (one for each memory unit), while the other synchronisations are performed by channels k (one for each pair of communicating components).

A periodic sensor is illustrated in Figure 4(a). It first performs the initial task[2], which lasts between e_1 and e'_1 time units. Then it starts a loop composed of a periodic data acquisition represented by a task at location T, which lasts between e_2 and e'_2 time units, and which is followed by a series

[2]which may correspond to the initialisation of sensor parameters

of synchronisations with target PUs in order to communicate the observed data. These synchronisations are performed in the order specified by the target list (or by the components declaration order for those that are not in the target list). Notice that it may happen that the cycle of a periodic sensor has an unbounded duration due to an impossibility to synchronise with some target PU.

An aperiodic sensor (see Figure 4(b)) has a similar shape, except that it does not have a separate initialisation phase. The guard ascertains that there is a minimal delay of e time units between two events.

A First processing unit may have one or more inputs, coming from sensors or processing units. It starts processing when data are received on one of its inputs. Figure 4(c) depicts a TAST model of $\mathsf{First}(k_1[e_1, e_1'], k_2[e_2, e_2']) \rightarrow (M_1, M_2)$ that inputs data from components k_1 or k_2 and processes them in the time intervals $[e_1, e_1']$ in location $P1$ and $[e_2, e_2']$ in location $P2$, respectively, and writes data to memory M_1 with a processing interval $[e_3, e_3']$ in location T_1, and to memory M_2 with processing $[e_4, e_4']$ in location T_2. Figure 4(d) depicts a similar First PU, simplified when the processings of data from k_1 and k_2 are the same.

A Both processing unit has two inputs and starts processing when both input data are received (in either order). Figure 4(e) depicts a TAST model of $\mathsf{Both}(k_1, k_2)[e_1, e_1'] \rightarrow (M_1)$ that awaits data from k_1 and k_2, and starts when both are available. The processing duration is in time interval $[e_1, e_1']$ in location P, after which data is written into memory M_1 in location T, taking a delay in $[e_2, e_2']$.

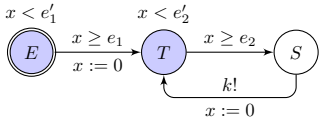
A Priority processing unit proposes a choice between two behaviours: if the input from k_m (master) comes first, then a processing with a duration in time interval $[e_1, e_1']$ is launched in location P, otherwise k_s (slave) comes first, awaits k_m and start a processing with duration in $[e_2, e_2']$ launched in location P_s. If both k_m and k_s are present at the same time, the choice is made non-deterministically. A TAST model of $\mathsf{Priority}(k_1[e_1, e_1'], k_2[e_2, e_2']) \rightarrow (k)$ is depicted in Figure 4(f). Again, like for a First processing unit, the schema may be simplified when the timing of the processing of the read data is the same with or without the slave input.

A shared Memory unit is a simple lock/unlock loop, as illustrated in Figure 4(g) (the time intervals indicated in the source list are managed in the automata of the sources).

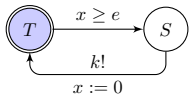
A Rendering loop is a cycling TAST reading a memory, processing the read data within some time interval, and producing a rendering, which takes again some time: this is illustrated in Figure 4(h).

Notice that there are two kinds of communications: All the communications with a memory unit use a pair of lock/unlock (with a processing in the processing unit or rendering loop using the memory unit). On the contrary, each communication to a processing unit uses a separate channel.

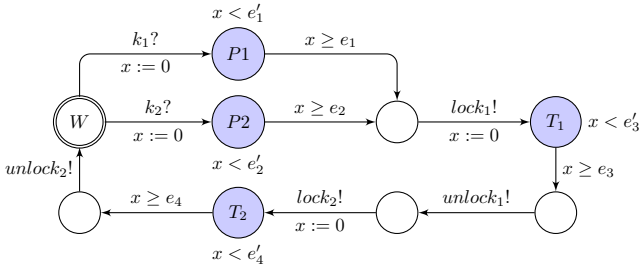
It may be observed that those models do not use auxiliary clocks. We left them as a possibility in the TAST syntax however, because it could happen that additional components are introduced in MIRELA at some point, where for example an activity is split into two sub-activities, with a constraint on



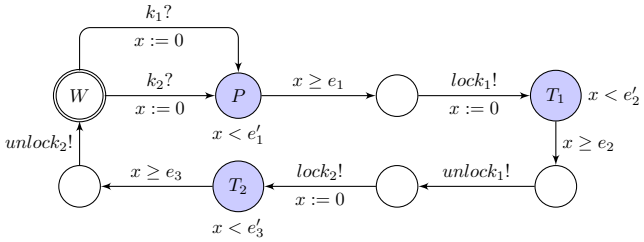
(a) A periodic sensor with (only one) output k



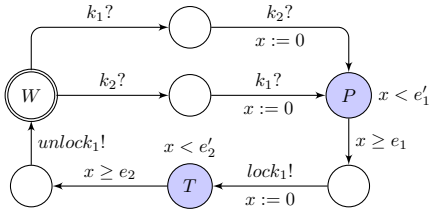
(b) An aperiodic sensor with output k



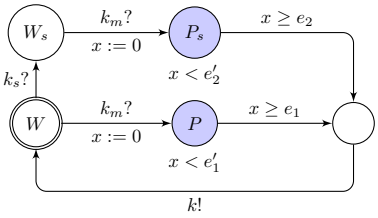
(c) A First PU with inputs k_1 and k_2, writing to memories M_1 and M_2



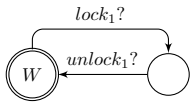
(d) A First PU with the same processing for inputs k_1 and k_2



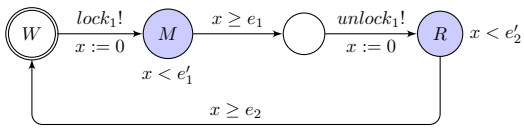
(e) A Both PU with inputs k_1 and k_2, writing to memory M_1



(f) A Priority PU with inputs k_m (master) and k_s (slave), and a single output k



(g) Memory M_1



(h) A Rendering loop reading from memory M_1

Fig. 4. TAST models of MIRELA components.

the sum of the times spent for them. Moreover, some clocks may later be added to measure the time taken to travel between two locations.

The MIRELA to UPPAAL gateway has been automated by developing a compiler using a parametric approach [4]. For Examples 1, 2 and 3, this leads to the models illustrated in Figures 5, 6 and 7, respectively.

Let us detail the behaviour of Example 1 modelled in Figure 5. One may observe first the difference between the behaviour of aperiodic and periodic sensors, respectively G and C. The former is modelled by a simple loop without any invariant, which does not guarantee that it will eventually leave locality G, while the latter is modelled by an initialised loop with invariant at location C ensuring that location C will be left before at most 450 time units. Data from sensors is communicated to Pr through the synchronisations on channels k_C and k_G. In order to travel from location W to P, Pr must synchronise on k_C while the synchronisation on k_G is optional. In both cases, the duration of the task at P is assumed to be the same (between 250 and 350 time units). Then, Pr makes a lock on memory M by synchronising on channel $lock$, writes data while at location M (between 20 and 30 time units) and releases memory by synchronising on $unlock$. Rendering R accesses the memory in the same way using $lock/unlock$ pairs while competing with Pr. As a consequence the durations of cycles in Pr and R (but also in C and G) depend on this competition.

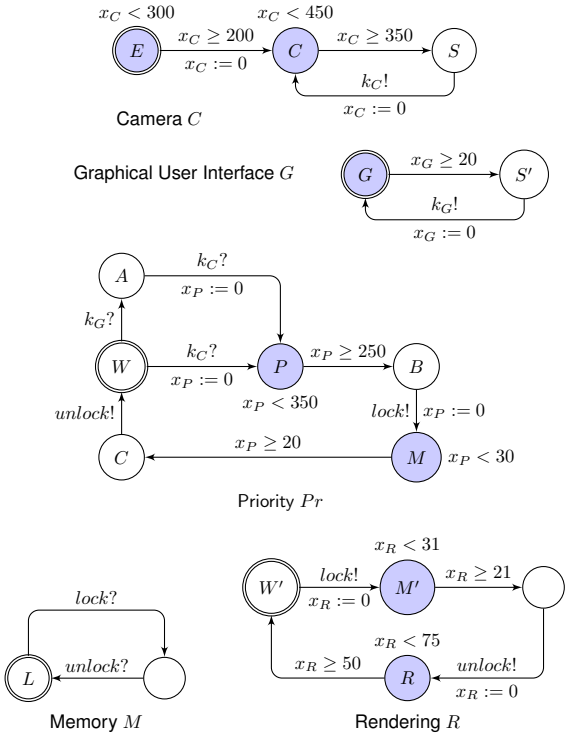


Fig. 5. TAST specification $\mathcal{A}$ for Example 1.

V. Analysis

A. Bad behaviours

Various kinds of deadlocks, or other usually considered highly harmful behaviours, may be distinguished for timed

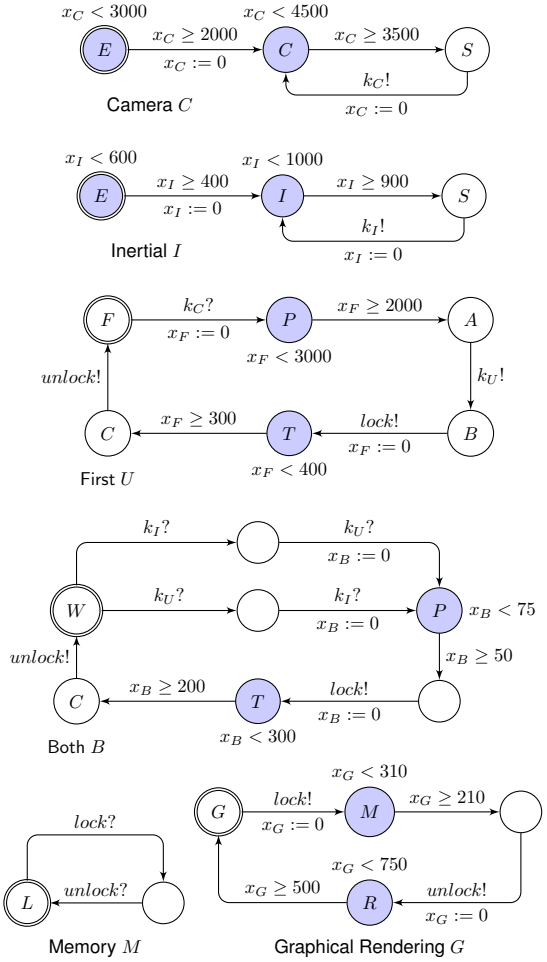


Fig. 6. TAST representations for Example 2.

automata:

- 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 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 disagreeable situation occurs when a location change is available after some time but this waiting time is unbounded. Moreover, for a network of timed automata (hence for TAST systems) we can distinguish:

- a *local deadlock* that occurs if no location change is available for some component while other components may evolve normally;

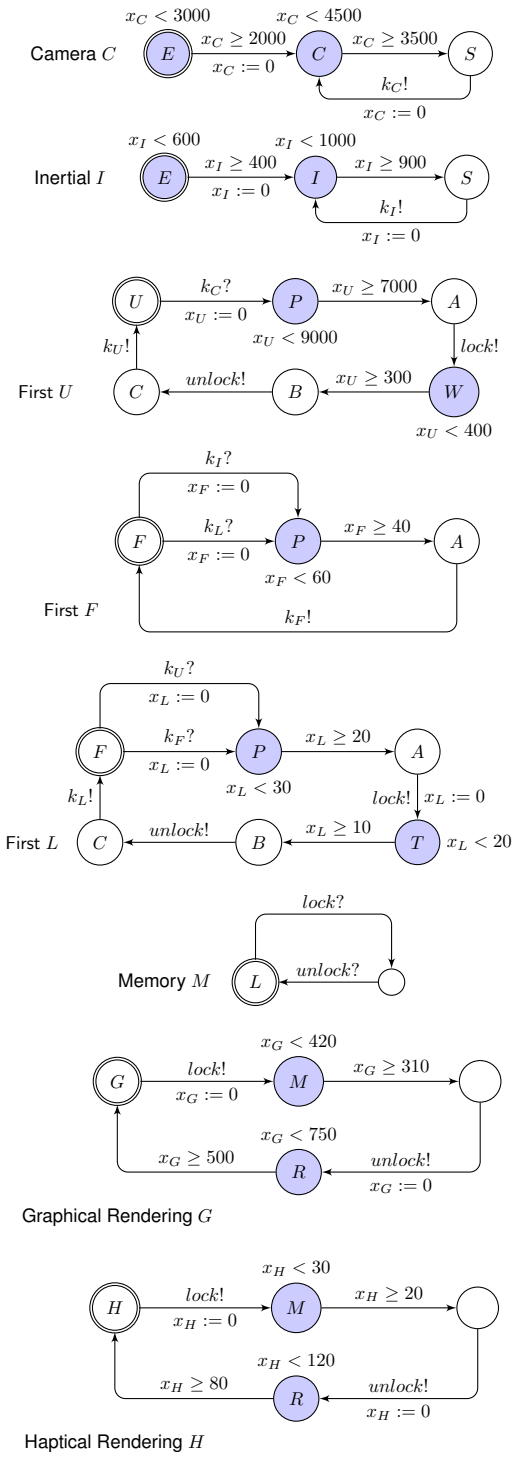


Fig. 7. TAST representations for Example 3.

- a *local unbounded waiting* that occurs if a component may evolve but the time before a change is unbounded (usually called a *starvation* situation). Notice however that this phenomenon is not always to be avoided, for instance if the component corresponds to a failure handling.

B. *Deadlocks in* MIRELA

We may first observe that:

Proposition 1: No (strong or weak) Zeno situation may happen in a MIRELA system.

Proof: The property holds since each loop in a component:

- either contains an arc with a guard $x \geq e$ (with $e > 0$) and a reset on some clock x, so that it is impossible to complete it in a null time; as a consequence the loop may be completed only a finite number of times in a finite delay;
- or contains only arcs with input communications (such as in a memory cell for instance); as a consequence it may only progress indefinitely while communicating with a loop of the previous kind.

□ 1

Concerning deadlocks, we may first observe the following:

Proposition 2: Let MS be a MIRELA system and TA its corresponding network of timed automata.

1) a component may only deadlock in a waiting location;
2) a memory unit may only deadlock if all its users deadlock elsewhere;
3) a rendering loop may not deadlock;
4) a system with a rendering loop may not have a global deadlock.

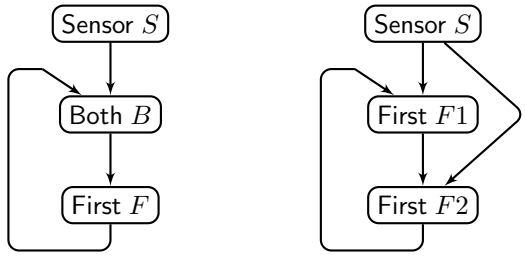
Proof:

1) If MS is well-formed, i.e., all components are correctly specified and in each $min - max$ pair, $min < max$, in TA each activity location has an invariant of the kind $x < max$ (or empty), x is reset in each incoming arc, and the unique output arc has a guard $x \geq min$. Hence, when x reaches min, the transition may occur since the guard of the next location may not be false. If the transition does not occur at that time, it remains enabled until x reaches max if the latter is finite, in which case it must occur.
2) In a memory unit, if a lock is performed by a component, it is certain from the semantics of the component that the unlock will be performed after some (bounded) time. Hence it may only be blocked if no user is able to perform the lock, i.e., if they are all deadlocked at a waiting location (see the previous point) different from a memory access request.
3) In a rendering loop the only waiting locations correspond to a memory access, and the property results from the previous point.
4) This is a direct corollary of the previous point.

□ 2

From the previous results, a global deadlock may not occur in a complete system, i.e., having at least one memory unit and an associated rendering loop, while a starvation may occur if the memory is continually used by other units, and no fairness strategy is applied. Moreover, local deadlocks may occur. Let us consider for instance the (simplified, without memory and rendering) system illustrated in Figure 8(a): after receiving a

signal from (periodic) sensor S, the processing unit B waits for a signal from processing unit F, but the latter waits for a signal coming from B. We thus have a deadlock of type $\mathcal{R}$ (for *reception*), where a group of processing units are each waiting for a signal reception but this signal may only be emitted by another processing unit of the group. Such a group may be decomposed in one or more cycles of "reception from a unit in reception state". Once such a deadlock occurs, more components may enter a local deadlock state: for instance, in the deadlock of type $\mathcal{R}$ occurring in Figure 8(a), the sensor S will soon join the set of deadlocked components since its next emission will not be absorbed by the deadlocked processing unit B.



(a) Deadlock 1 (b) Deadlock 2

Fig. 8. Examples of (local) deadlocks. Renderings and memories are omitted.

Another (symmetrical) kind of local deadlock is illustrated in Figure 8(b): when sensor S emits signals to processing units $F1$ and $F2$, they both process it, then each one tries to emit a signal to the other one, which cannot succeed since they are waiting for emission and may not perform a reception. This is a deadlock of type $\mathcal{E}$ (for *emission*), where processing units in a group are each waiting to emit a signal, but that signal may only be received by another processing unit of the group, thus forming one or more loops of "emission to a unit in an emission state". Again, once such a deadlock occurs, more components may enter a deadlock state: for instance, in the deadlock of type $\mathcal{E}$ occurring in Figure 8(b), the sensor S will join the set of deadlocked components since its next emission will not be absorbed by the deadlocked processing unit $F1$ (or $F2$, depending on to which unit its signals are first sent).

These two kinds of local deadlocks 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.

But there are also deadlocks of a mixed nature, combining components in emission and reception phases, like the situation illustrated in Figure 9: assume that, first, sensor $S1$ sends a signal to $F1$, which accepts it and sends a signal to B which also accepts it; then $S1$ sends a new signal to $F1$, which accepts it and sends another signal to B, which cannot accept it at this stage since the latter waits for a signal from $F2$; now, if Sensor $S2$ sends a signal to $F2$, which accepts it and the latter tries to send a signal to $F1$ (before trying to send a signal to B): $F2$ is blocked since $F1$ waits for its second signal to B to be accepted, but B itself waits for a signal from $F2$, which cannot happen since the latter is presently waiting for its signal to $F1$ be accepted. Note that the same happens if the emissions by $F2$ are produced in the reverse order, but the trace leading to a blocking situation is a bit longer.

A similar but even more intricate situation occurs if we use in the same example a Priority processing unit instead of Both B. Actually, if the input from $F2$ is considered as master in B, then the system presents a deadlock, which is not the case if the input from $F1$ to B is a master. More importantly, these deadlocks do not involve a cycle of communicating (sensor or processing) units, unlike what happened in the $\mathcal{E}$ or $\mathcal{R}$ cases. This is due to the fact that, when a component deadlocks 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 deadlocked components) to a non cyclic flow of information.

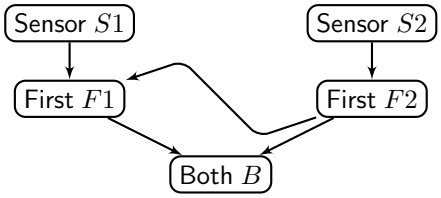


Fig. 9. Example of a (local) deadlock without a loop of Processing units.

In order to derive guidelines for an efficient detection of possible deadlocks, we first perform the following observations. If MS is a MIRELA system, let us denote by MS' the system obtained by dropping all rendering loops and all time constraints (clocks, guards and invariants). Dropping rendering loops is easy since this has no impact on the other components of the system. And let MS'' be MS' where all memory units are also dropped; this is slightly more involved since some processing units use explicitly some memories: in the corresponding timed automata, one has to remove the $lock!/unlock!$ labels from the arcs.

Proposition 3: Let MS be a MIRELA system:

1) if a local deadlock is present in MS, it is also present in MS', as in some connected subsystem of MS'';
2) if a component deadlocks in MS' or 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.

Proof:

1) From Proposition 2(2-3), if there is a (local) deadlock there is a deadlocked component which is not a rendering loop nor a memory unit. From Proposition 2(1), that component is blocked in a waiting location. Then it is also blocked, after some finite time, if we enlarge some or all intervals in MS, since this allows to reach the same situation and the group of components blocking each others does not depend on timing constraints (deadlocks are simply due to mutual waits). In particular the deadlock remains if we replace each min by 0 (or equivalently if we remove all guards). Moreover, now it is also possible to reach the deadlock by shrinking the time, allowing

to also replace all max by a same arbitrary value d (or by ∞, which amounts to remove all invariants). Finally, the rendering loops and memory components may be dropped, since their only possible effect on the evolution leading to the deadlock is to delay some components by some finite time, but the same effect may be obtained by increasing the time used by those components to handle the corresponding data. Hence if there is a local deadlock in MS, the same situation will occur in MS', as well as in some connected subsystem of MS''.

2) This results from a quick analysis of the propagation of deadlocks. Note that if a component deadlocks while it is a source of a slave input to a Priority, the latter does not necessary deadlock; 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. Otherwise, it does.

□ 3

These results are precious since the automatic detection of deadlocks (with UPPAAL for example) is especially difficult when the system is complex, when timing constraints exhibit different orders of magnitude, when there is no deadlock (because it is then necessary to explore the whole state space) and when local deadlocks do not propagate to the entire system.

Hence we propose the following methodology to detect the presence or absence of deadlocks:

1) First detect if there is a (global or local) deadlock in MS' or in some connected subsystem of MS''. This is much more efficient than doing the same directly on MS, since the systems to be considered are much simpler, there is no timing constraints and there is some hope that local deadlocks will become global ones. If there is no such deadlock, from Proposition 3 we may deduce MS is deadlock free. If a deadlock is detected, it is usually simple to check if the same situation is also reachable in the original system MS.
2) A global deadlock may be checked on MS' or each connected subsystem of MS'' with the UPPAAL query `A[] not deadlock`; this is enough if it is sure that local deadlocks propagate to the entire subsystem (see Proposition 3-2). Unfortunately, a local deadlock (not propagating to a global one) is much more delicate to detect in MS' or in a connected subsystem of MS'': this would need the usage of imbricated queries to check that, for any reachable state, if a component is in a waiting locality, it is possible to leave the latter, and UPPAAL does not allow them.
3) If the previous checks are non-conclusive, a local deadlock in MS may be searched for with the aid of liveness queries of the kind `C.L --> C.N`, checking that, in a component `C`, if we reach a location `L` it is certain that we shall eventually reach a location `N` afterwords; this may be used to check that a component is not stuck in a waiting location; however, besides the time needed for such queries (which may be prohibitive), a further analysis may be necessary to avoid a confusion with an (expected) unbounded waiting (see next section).

Table I illustrates some applications of this methodology, using Intel(R) Quadri-Core(TM) i5-2400 CPU at 3.10GHz and 8GB of RAM. Lines 1 and 2 show that no global deadlock is found in the simplified versions of Examples 1 and 2, while Proposition 3-2 shows that a local deadlock would lead to a global one in those cases; hence those examples are deadlock-free. On the contrary, line 3 shows that the simplified version of Example 3 presents a global deadlock, and it is easy to see that the same situation remains reachable in the original version. By contrast, a direct analysis of this example takes hours to detect that component I may not always perform a full loop (see lines 4-5), and the application of acceleration features of UPPAAL (called -Z and -A, lines 6-7) are non-conclusive.

TABLE I. RESULTS OF CHECKING FOR (LOCAL) DEADLOCKS

	case	query	time	# states	result
1	Ex'_1	A[] not deadlock	1ms	28	TRUE
2	Ex'_2	A[] not deadlock	2ms	175	TRUE
3	Ex'_3	A[] not deadlock	5ms	500	FALSE
4	Ex_3	I.I --> I.S	21h38'	36 009 471	TRUE
5	Ex_3	I.S --> I.I	11'10s	710 921	FALSE
6	Ex_3 (-Z)	I.I --> I.S	15'55s	80 145 889	MAY BE
7	Ex_3 (-A)	I.I --> I.S	11s	247	MAY NOT

C. *Unbounded waitings in* MIRELA

Besides true deadlocks, situations may arise where a component may undergo an unbounded 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 avoid to use them in a Both unit, as a master to a Priority, or in a First when there is no other kind of input.

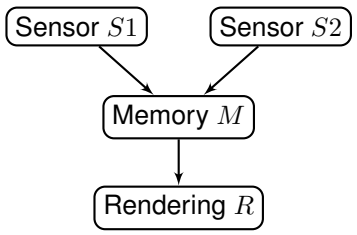
Another kind of situation occurs if several components compete to communicate and, due to the non-deterministic way choices are performed, some of them never succeed. In a MIRELA system, this may for instance occur when performing a lock on a memory unit (note that unlocks may only be performed by the components having succeeded in the lock): it may happen that a component $Comp$ 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 $Comp$ is never chosen, unfortunately, again and again. This also shows that here we should not consider simplified systems, without memory and rendering units, since the latter ones may be essential ingredients for inducing unbounded waitings.

This may be observed, for instance, in Example 4 (cf. Figure 10), where sensors $S1$ and $S2$ can alternately write in Memory M while Rendering R may wait indefinitely. Actually, it happens for example if $S1$ performs its task while $S2$ is writing in M allowing $S1$ to be ready to access it when $S2$ releases the lock, and they continue to indefinitely access M alternately. This behaviour may occur if the maximum writing time (in red) of a sensor to M is strictly greater than

the minimum time (in blue) needed by the other sensor to perform its task. Here, the maximum writing time of $S1$ is 250, meaning that if we change the definition of $S2$ to

$$S2 \quad = \quad \mathsf{Periodic}(400, 600)[250, 500];$$

the starvation no longer occurs, because when $S1$ releases the lock on the memory, $S2$ is still executing its task and so, R is the unique competitor to the memory and thus has to access it (since the communication is urgent). In other words, this case of starvation of R depends only on the time constraints of $S1$ and $S2$ but not on those of R itself. The time constraints of R may only impact the number of memory accesses R may perform before the starvation occurs.



Ex_4 :

$$\begin{array}{rcl} S1 & = & \mathsf{Aperiodic}(300); \\ S2 & = & \mathsf{Periodic}(400, 600)[200, 400]; \\ M & = & \mathsf{Memory}(S1[200, 250], S2[250, 350]); \\ R & = & \mathsf{Rendering}(50, 100)(M[20, 30]). \end{array}$$

Fig. 10. Scheme and specification of Example 4.

The same kind of behaviour may happen when a component *Comp* tries to emit to a First unit and the latter allows several inputs: if at least one other component is ready to emit whenever the First component enters the location at which it may receive an input (actually W), it may happen that, unfortunately, *Comp* is never chosen. An analogous situation may occur if a component *Comp* tries to synchronise as a slave with a Priority unit: if the master is also ready whenever the Priority component enters its location W, it may happen that, unfortunately, the slave *Comp* is never chosen.

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. This may again be checked using UPPAAL. Notice that the verification may be performed using the same kind of queries `C.L --> C.N` as for deadlocks, but the semantical information we have about the model helps us to distinguish between a deadlock and an unbounded waiting. In particular, locations potentially allowing unbounded waiting are known in the model (e.g., in an aperiodic sensor as in Example 1).

D. Temporal properties

Another kind of question that may be asked on such systems concerns minimal and/or maximal durations taken by components to perform operations. Average, median or percentile values are irrelevant since our systems are non-deterministic, but not probabilistic. 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. A typical query to do that is `sup{C.L}:C.x`, which determines the supremum of the clock x in location L for component C.

TABLE II. RESULTS OF CHECKING FOR DURATIONS.

	case	query	time	# states	result
1	Ex_2	sup{G.R}: G.y	10ms	1 049	< 1 760
2	Ex_2	sup{B.C}: B.y	60ms	4 650	< 11 510
3	Ex_2	sup{U.C && !U.firstCycle}: U.y	30ms	3 315	< 6 210
4	Ex_2	sup{I.S}: I.xi	70ms	6 795	< 10 210
5	Ex_2	sup{C.S}: C.xc	0ms	812	< 4 500

Usually, the cycle rates of a component are impacted by the rest of the system, and the complete information is difficult to obtain before the implementation. Using the MIRELA approach allows to overcome this difficulty by performing analyses on the model (so before any implementation). Table II shows the results of measures concerning the actual maximal duration of the cycles of various components, allowing to assess the actual "cadence" of the system evolution. To do so, new clocks were introduced in some automata, like y in lines 1-3, which is reset at the beginning of a cycle. For the property in line 3, we used a Boolean variable *firstCycle* initially set to true and switched to false when the camera C gets its first image; this allows to get rid of the initial phase, when the camera has not yet reached its own cycle.

For the rendering loop R (line 1), we obtain a duration that is longer than the defined rendering task and the corresponding memory reading (1760 time units instead of $1060 = 310+750$) showing that it may have to wait for the other components of the system, i.e., B and U, which also compete to access the memory. We obtain similar results for the maximal actual duration of the cycle of each processing unit (B and U) in lines 2 and 3, and sensors (C and I) in lines 4 and 5, showing that they are all impacted by the rest of the system.

VI. TOWARDS A CORRECTION METHODOLOGY

Of course, it would be interesting to develop techniques to overcome the detected problems, when there are some. In order to illustrate what can be done, we considered the local deadlock in Example 3. In this example, the First components L and F exhibit a local deadlock because they have to communicate with each other, but if they initially get signals from I and U they will both be in emitting state, hence unable to exchange information (this leads to a deadlock of type $\mathcal{E}$ (emission)); then we should manage so that, when one is trying to emit a signal, the other one must be ready to receive it.

Next, we may observe that the semantics of this exchange of information may be interpreted as follows: when no information arrives from the outside world (from I for F and from C through U for L), F and L cooperate to get progressively a better perception of the current situation. But if some new information arrives from I and/or C, they have to take it in consideration, of course.

Now, in order to get this kind of behaviour, instead of using First components, one may use Priority ones instead, with the communications from the outside world as slaves: the details of the behaviour will be slightly different, but the general spirit

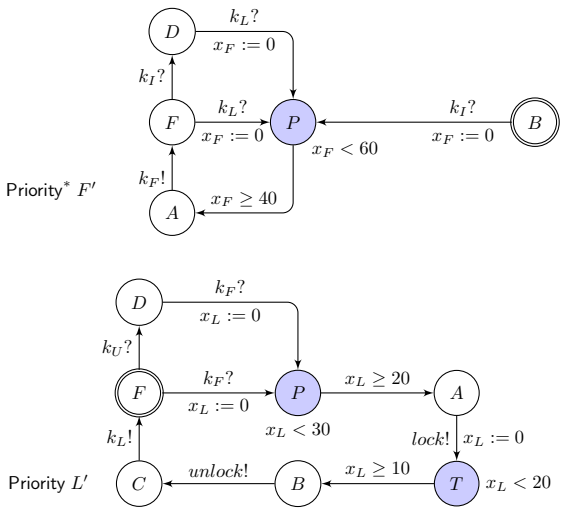


Fig. 11. Components Priority* F' and Priority L' replacing respectively components First F and First L in the corrected Example 3.

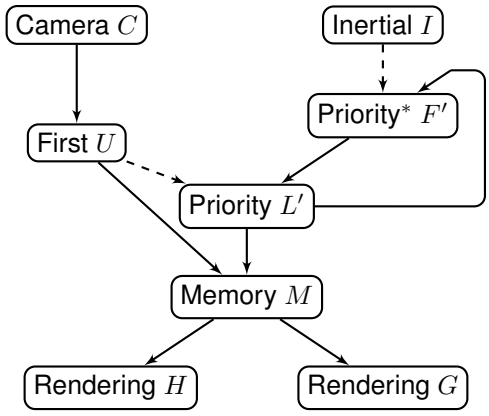
will be preserved. However, with the Priority components we introduced before, there is a serious problem: initially, the two components F and L will wait for each other in a $\mathcal{R}$-type deadlock. The idea to get out of this trap is to break the symmetry and introduce for one the those components a new kind of Priority, that we shall call Priority$*$, where initially one waits for the slave, handles it, and only after that we enter the normal loop of a Priority component. We shall use this here for the component closer to the outside world, i.e., F, but a similar behaviour would be obtained by choosing L instead. We then obtain the same system for Example 3 as before except that F is replaced by F', and L by L', as illustrated in Figure 11.

So, initially, F' will wait for a signal from I and then engage a communication with L as an emitter, while L' (after possibly receiving an information from U) engages a communication with F' as a receptor. Then, each time a communication is performed between F' and L', they will exchange their role, as required for a correct communication.

As expected (lines 1-2 in Table III), the system of Example 3 corrected as explained (cf. Figure 12), simplified as before for easing and speeding the automatic checks, no longer presents any deadlock. In general, that type of modification can introduce unbounded waitings, however, and this should be checked (lines 3-4) like in section V-C. (However, checking these properties on the unmodified system takes a long time, actually more than one day. Therefore, we checked them on a system where rendering loops were removed.) It seems that this kind of correction may be performed in many similar cases, but if a First component with many inputs is concerned, it may happen that we need to introduce Priority and Priority* components with more than one slave. Note also that if C would be replaced by an Aperiodic sensor, we would have a situation where C and U could never start because neither L nor M would require an input from them in order to work.

VII. Conclusion

We presented a fragment of the MIRELA framework for analysing and understanding time related issues in MR applications, focusing on the specification and (semi-) automatic



$$
\begin{aligned}
corEx_3: \\
C &= \mathsf{Periodic}(2000,3000)[3500,4500];\\
I &= \mathsf{Periodic}(400,600)[900,1000];\\
U &= \mathsf{First}(C[7000,9000])\rightarrow(M,L);\\
F' &= \mathsf{Priority}^*(L',I[40,60]);\\
L' &= \mathsf{Priority}(F',U[20,30])\rightarrow(M,F');\\
M &= \mathsf{Memory}(U[300,400],L'[10,20]);\\
G &= \mathsf{Rendering}(500,750)(M[310,420])\\
H &= \mathsf{Rendering}(80,120)(M[20,30]).
\end{aligned}
$$

Fig. 12. Scheme and specification of Example 3 corrected.

TABLE III. Results of checking in corrected Example 3.

	case	query	time	# states	result
1	$corEx_3'$	A[] not deadlock	8ms	503	TRUE
2	$corEx_3''$	A[] not deadlock	5ms	535	TRUE
3	$corEx_3$ wo G,H	I.I --> I.S	36'57s	1 519 043	TRUE
4	$corEx_3$ wo G,H	I.S --> I.I	9'24s	503 306	TRUE

verification of deadlocks and other important properties. A simple declarative, domain specific language has been defined in order to easily describe MR systems. It has been provided with a compiler generating the corresponding timed automata, and a verification methodology has been proposed. Such a checked specification may now be used in the MIRELA framework to produce an implementation skeleton aiming at preserving those properties (see [7]).

The approach has been illustrated with several small but realistic case studies, highlighting in some situations unexpected behaviours or errors. Various kinds of properties have been checked with UPPAAL and a correction methodology has been proposed for removing a deadlock in a cycling specification.

The verification of some temporal properties, like actual cycle or task durations, was unsatisfactorily long because of the usual state space explosion problem. We are currently developing suitable series of abstractions, consisting in grouping some automata and replacing them by tailored tasks, in order to allow to overcome this difficulty.

Acknowledgment

The authors are grateful to the anonymous referees, whose interesting comments allowed to improve the present paper. This work has been partly supported by French ANR project Synbiotic and Polish-French project Polonium.

REFERENCES

[1] Rajeev Alur and David 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] Rajeev Alur and David L. Dill. *The theory of timed automata*. In *Real Time: Theory in Practice (REX Workshop)*, volume 600 of *LNCS*, pages 45–73, 1991.

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

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

[5] Martin Bauer, Bernd Bruegge, Gudrun Klinker, Asa MacWilliams, Thomas Reicher, Stephan Riss, Christian Sandor, and Martin Wagner. *Design of a component-based augmented reality framework*. In *Proceedings of the International Symposium on Augmented Reality (ISAR)*, 2001.

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

[7] Raymond Devillers, Jean-Yves Didier, and Hanna Klaudel. *Implementing timed automata specifications: The" sandwich" approach*. In *Application of Concurrency to System Design (ACSD), 2013 13th International Conference on*. IEEE, pages 226–235, 2013.

[8] Jean-Yves Didier, Bachir Djafri, and Hanna Klaudel. *The* MIRELA *framework: modeling and analyzing mixed reality applications using timed automata*. In *Journal of Virtual Reality and Broadcasting*, 6(1), t urn:nbn:de:0009-6-17423,, ISSN 1860-2037, 2009..

[9] Jean-Yves Didier, Hanna Klaudel, Mathieu Moine, and Raymond 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] Christoph Endres, Andreas Butz, and Asa MacWilliams. *A survey of software infrastructures and frameworks for ubiquitous computing*. In *Mobile Information Systems Journal*, 1(1):41–80, 2005.

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

[12] Pablo Figueroa, J Hoover, and Pierre Boulanger. *Intml concepts*. University of Alberta. Computing Science Department, Tech. Rep, 2004.

[13] Michael Haller, Jürgen Zauner, Werner Hartmann, and Thomas Luckeneder. *A generic framework for a training application based on mixed reality*. Technical report, Upper Austria University of Applied Sciences, Hagenberg, Austria, 2003.

[14] Charles E Hughes, Christopher B Stapleton, Darin E Hughes, and Eileen M Smith. Mixed reality in education, entertainment, and training. *Computer Graphics and Applications, IEEE*, 25(6):24–30, 2005.

[15] Marc Erich 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, 2002.

[16] Mathieu Moine. *Implementation tool of timed automata specifications*. Master's thesis, ENSIIE – Université d'Evry-val d'Essonne, 2013.

[17] David Navarre, Philippe Palanque, Rémi Bastide, Amelie Schyn, Marco Winckler, Luciana P Nedel, and Carla MDS Freitas. *A formal description of multimodal interaction techniques for immersive virtual reality applications*. In *Human-Computer Interaction-INTERACT 2005*. Springer, pages 170–183 2005.

[18] Wayne Piekarski and Bruce H. Thomas. *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*, Washington, DC, USA (IEEE Computer Society), page 247, 2003.

[19] G. Reitmayr and Dieter Schmalstieg. *An open software architecture for virtual reality interaction*. In *Proceedings of the ACM symposium on Virtual reality software and technology*. ACM Press, pages 47–54, 2001.

[20] Christian Sandor, Thomas Reicher, et al. *Cuiml: A language for the generation of multimodal human-computer interfaces*. In *Proceedings of the European UIML conference*, volume 124, 2001.

[21] Uppaal. http://www.uppaal.org/.

[22] Md Tawhid Bin Waez, Juergen Dingel, and Karen Rudie. *Timed automata for the development of real-time systems*. Research Report 2011-579, Queen's University – School of Computing, Canada, 2011.