Transforming CCSL partially-ordered Traces into UML Interaction Diagrams

Kelly Garcés, Julien Deantoni, Frédéric Mallet

▶ To cite this version:

Kelly Garcés, Julien Deantoni, Frédéric Mallet. Transforming CCSL partially-ordered Traces into UML Interaction Diagrams. [Research Report] RR-7842, INRIA. 2011. hal-00652987

HAL Id: hal-00652987

https://hal.archives-ouvertes.fr/hal-00652987

Submitted on 16 Dec 2011

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Transforming CCSL partially-ordered Traces into UML Interaction Diagrams

Kelly Garcés — Julien DeAntoni — Frédéric Mallet

N° 7842

Décembre 2011

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7842--FR+ENG

Transforming CCSL partially-ordered Traces into UML Interaction Diagrams

Kelly Garcés , Julien DeAntoni , Frédéric Mallet

Thème COM — Systèmes communicants
Projet AOSTE

Rapport de recherche n° 7842 — Décembre 2011 — 28 pages

Abstract: The need for verification and debugging of critical temporal constraints in embedded systems comes out at different stages of development. In the specification step, static and dynamic views of the system are established and simulations are performed. In the implementation step, code may be instrumented with the purpose of collecting traces as the system executes in a target platform. In the same fashion as system executions, simulations produce traces that are later on analyzed by means of textual scripts. Instead of intricate scripts, we believe that the use of visual artifacts such as UML interaction diagrams (*i.e.,* sequence and timing diagrams) can ease the comprehension of system behavior. In this report, we propose partial orderings (which order the events reported in traces in a temporal and causal way) as a pivot to go toward interaction diagrams straightforwardly. Mappings between partial orderings and UML interaction diagrams are implemented as transformations. We illustrate our approach with a prototype and an example.

Key-words: UML, Traces, Logical time, Debugging, Verification, Model-Driven Engineering

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Une approche de transformation de modèles pour dériver des diagrammes d'interaction UML à partir des traces CCSL partiellement ordonnées

Résumé : Le besoin de vérification et débogage des constraintes de temps réel d'un système embarqué apparaissent à de différents stades d'un développement. Dans l'étape de spécification, des vues statiques et dynamiques du système sont établies et des simulations sont effectuées. Dans l'étape de mise en oeuvre, le code peut être instrumenté dans le but de recueillir des traces du système lors de son exécution dans une plate-forme cible. De la même façon que les exécutions du système, les simulations produisent des traces qui sont ensuite analysées en utilisant des scripts textuelles. Au lieu des scripts complexes, nous croyons que l'utilisation d'artefacts visuels tels que les diagrammes d'interaction UML (*i.e.*, diagrammes de séquence et de temps) peuvent faciliter la compréhension du comportement du système. Dans ce rapport, nous proposons l'ordre partiel (lequel ordonne les événements consignés dans les traces d'une manière temporelle et causale) comme un pivot pour aller vers des diagrammes d'interaction. Les alignements entre l'ordre partiel et les diagrammes d'interaction UML sont implémentés comme des transformations. Nous illustrons notre approche avec un prototype et un exemple.

Mots-clés : UML, Traces, Temps logique, Débogage, Vérification, Ingénierie Dirigée par les Modèles

1 Introduction

UML MARTE is one of the common used approaches to implement MDE for real-time and embedded systems. Functional and extra-functional aspects are expressed in models which undergo a set of transformations that finally results in executable code.

Since the code may be deployed over heterogeneous, communicating computational units, the order of incoming events and their frequency are often unpredictable. Academia and industry are aware of that, while they are interested on using MDE to obtain a high-abstraction of a system, they also investigate how to verify/debug, at model level, its behavior and functioning as simulations or executions of such a system happen. A low-impact verification technique consists of collecting traces (i.e., logs) of program segments as the software runs on target, and then analyzing offline such traces by means of scripts. Scripts are time consuming to produce and result in difficult-to-read specifications [1].

Instead of intricate specifications, we focus on the use of UML interaction diagrams (in particular sequence and timing diagrams) to leverage the comprehension of system simulations or executions. Sequence and timing diagrams have different purposes: whereas the former concentrates upon the interchange of messages between lifelines, the latter allows reasoning about time and the conditions that change lifelines.

We propose mappings that allow us to go toward these two kinds of diagrams from static and dynamic specifications and partial orderings[1]. UML class and composite structure diagrams act as a static specification and CCSL (Clock Constraint Specification Language) [2] as a dynamic specification. Partial orderings are traces plus the temporal and causal relations between them. It worth mentioning that the output of our process are interaction diagrams that describe one of the possible scenarios (either valid or invalid) of simulation or execution of a system. Therefore, this output varies from classical interaction diagrams defined at design level which are intended to cover the set of valid scenarios.

We implement the mappings between partial orderings and UML interaction diagrams as transformations and we illustrate their applicability by means of an example. Our proof of concept demonstrates that: 1) partial orderings (along with static and dynamic specifications) are expressive enough to derive most of interaction diagram semantics and 2) resulting diagrams can be visualized in open-source tool editors.

Because the proposed transformations can take as input traces coming from simulations or executions, our approach supports verification and debugging, at model level, in early (*i.e.,* specification) or advanced (*i.e.,* implementation, deployment) stages of software development. In addition, the use of partial orderings give us a global vision of the system execution independently of whether the traces are isolated or not[2]. Finally, when the selection of only one kind of interaction diagram is essential, we use clock trees as visual artifacts that suggest the most appropriate diagram with regarding to the system nature, *i.e.,* synchronous or asynchronous.

[1] Partial ordering and partial order are used as interchangeable terms in this report.

[2] There is a need for isolation in offline debugging where trace storage produces overhead on the communications and then faulty executions. For these cases, we have proposed a reconciliation approach whose output is a global trace in the form of a partial order [3].

The report is structured as follows: Section 2 and Section 4 recall the basic semantics of UML sequence and timing diagrams and the CCSL language. Section 3 compares our approach to related work. Section 5 presents an example that is used to illustrate our approach described in Section 6. Section 7 concludes the report.

2 Background

In this section, we briefly describe the semantics for those aspects of UML interaction diagrams that we use in this report. We will assume that the reader is broadly familiar with UML sequence and timing diagrams.

Let us introduce the graphical notation of sequence diagrams by using Figure 1. Each vertical line describes a lifeline for an individual participant where time increases down the page. Messages define one specific kind of communication between lifelines. A communication can be, invoking an operation, creating or destroying a participant. Besides the kind of communication, messages specify the sender and the receiver. Messages are depicted by arrows whose style reflects particular properties, for example, open-head arrows represent asynchronous messages, filled-head arrows symbolize synchronous messages, dashed arrows describe reply messages. When a message invokes an operation, the execution of such an operation is represented as a thin white rectangle (referred to as execution specification in [4]). An execution specification has a starting point and a finishing point.

Events on the same lifeline are ordered linearly down the page (*i.e.,* they are totally ordered), except within a coregion or a parallel combined fragment where the order of event occurrences is insignificant. Whereas a coregion can only occur in a single lifeline, a parallel combined fragment merges the behavior of at least two lifelines. A coregion is depicted by short horizontal lines and a parallel combined fragment is delineated by a box labeled with the keyword PAR. Besides PAR, the OMG has defined other operators which enable the combination of fragments, for instance, ALT represents two possible choices or alternatives of behavior. Finally, destruction events, depicted by a cross in the form of an *X*, indicate that no other occurrence may appear below of it on a given lifeline.

Now we elaborate on the graphical notation of timing diagrams (see Figure 2). In a timing diagram, lifelines are shown in separated compartments arranged vertically. The horizontal axis indicates that time increases in ticks from left to right. The vertical axis can be labeled with a list of states or values. The state lifeline shows the change of state of an participant over time, whereas a value lifeline shows the change of value of an element over time [5]. Timing diagrams come with messages (raising signals) and destruction events.

From this description one concludes that sequence and timing diagrams mostly share concepts however they have slightly different capabilities that make them more appropriate for certain situations. Sequence diagrams focus on message exchange and timing diagrams are useful when the primary purpose of the diagram is to reason about time.

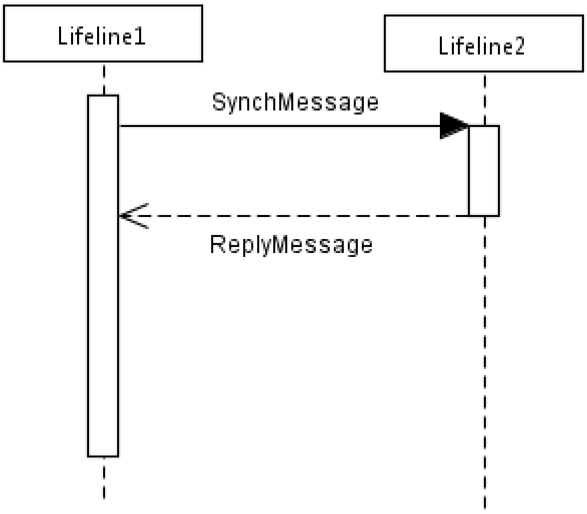

Figure 1: An example of UML sequence diagram

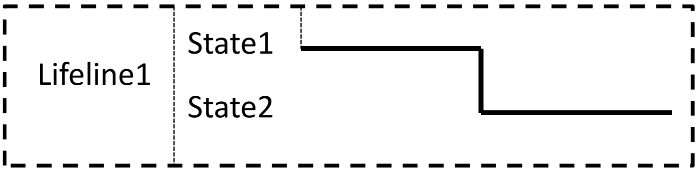

Figure 2: An example of UML timing diagram

3 Related work

From our bibliographical study, we have identified two directions in the matter of verification/debugging based on UML interaction diagrams. Former work [6][7][8] goes from interaction diagrams to more formal notations (such as timed automata or timed graphs) where one can take advantage of model-checking techniques to perform verification. A second direction, recently presented by Iyenghar et al. [9][10] and Apvrille et al. [11], is to go from traces to interaction diagrams which may be animated. The animation allows embedded software engineers to debug the system behavior as it is deployed in a target platform. Whereas Iyenghar et al. work with execution traces, Apvrille et al. use simulation traces.

This report is aligned with the second direction, let us compare our approach with Iyenghar and Apvrille works, which are the closest related works:

- In Iyenghar's work, each event happening in the target platform is immediately communicated to the host side which animates a sequence or timing diagram, it is referred to as animation at runtime. In our case, the diagrams are produced only after the whole trace data has been collected, that is, offline rendering.
- A consequence of animation at runtime is overhead. Although the delay introduced by Iyenghar's monitor routine is pointed out as negligible for the MIDI (Musical Instrumental Digital Interface) system case study, it might not be the case for other real-time systems where even small overhead can lead to faulty executions because of the tight synchronization constraints. Our approach, in turn, allows debugging while still minimizing the overhead.
- In Apvrille et al., software components are first modeled and formally verified using, respectively, a UML profile and UPPAAL [12]. Then code is generated from the resulting models and its execution on a target platform is simulated. The simulation output is a set of traces which can be displayed in the form of sequence diagrams during code execution or after.
- A disadvantage in Apvrille's work is the use of an ad-hoc UML profile so-called AVATAR. We, in turn, use UML MARTE which is widely supported by most commercial and open-source tool editors.
- Unlike Apvrille, which animates resulting sequence diagrams, we provide just a visualization of them. As a part of future work, we plan to animate sequence diagrams as we do with timing diagrams.
- Whilst Iyenghar et al. and Apvrille et al. assume that trace data is totally ordered, we use a partial order which gives us a global vision of the system behavior independently of whether traces come from isolated parts of the system. Therefore, our approach covers a more general case where the system may be built up from a number of heterogeneous and distributed pieces which are likely to be unsynchronized, or based on different forms of time references.

- Finally, our approach moves a step further than Iyenghar and Apvrille work since it derives interaction diagrams from both simulation and execution traces.

4 CCSL in a nutshell

The Clock Constraint Specification Language (CCSL) was initially defined as a companion language for the Time Model of the UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [13]. The central concept on MARTE Time model is the notion of *clock*, which represents a (possibly infinite) totally ordered set of *instants* [14]. In this model, clocks extend the Unified Modeling Language (UML) [15] events and instants stand for the event occurrences. These clocks can be logical or physical, dense or discrete. In the remainder of this paper we only consider discrete clocks, whether logical or physical.

The MARTE time model also provides *Clock Constraints* that refer to at least two clocks and constrain the respective evolution of their instants.

CCSL has a formal semantics [2] that can be exploited to detect invalid specifications (*e.g.,* deadlocks) or compute a correct execution (by simulation), if any, in the TimeSquare tool [16]. Foundational CCSL constraints are defined in a kernel library. CCSL allows building new libraries and the definition of user-defined constraints by composing existing relations (from the kernel library or from other libraries) in order to build specific constraints adequate for a given domain.

CCSL is a means for specifying constraints on the evolution of clocks. A constraint can be either a *relation* or an *expression*. A CCSL expression defines a new clock based on existing ones. In this paper, we do not give all the details about the semantics of CCSL but a full description is available as a research report [2]. However, we informally describe the relations and expressions used in this document.

We consider all the instants for a given system, $\mathcal{I}$, and we build a time structure $\langle \mathcal{I}, \prec, \equiv \rangle$ on it. $\prec$ is an irreflexive and transitive partial relation called *precedence*. $\equiv$ is a partial equivalence relation, *i.e.,* reflexive, transitive and symmetric, called *coincidence*. From these two relations, we build two more: *causality* (denoted $\preccurlyeq$) and *exclusion* (denoted ($\#$). Let a and b be logical clocks, when a causes b, then either $a \prec b$ or $a \equiv b$. When a and b are exclusive, then either $a \prec b$ or $b \prec a$.

A clock $c = \langle \mathcal{I}_c, \prec_c \rangle$ is such that $\mathcal{I}_c \subset \mathcal{I}$ and $\prec_c$ is a projection of $\prec$ over $\mathcal{I}_c$ and is a total order relation. If $\mathcal{I}_c$ is discrete (c is called a discrete clock), we denote $c[k]$ the k^{th} instant of c where $k \in \mathbb{N} \setminus \{0\}$.

Clock relations are a practical way to create infinitely many instant relations at once. For instance, the clock relation *Precedes* (denoted $\boxed{\prec}$) defines infinitely many instant relations of the kind *precedence*. $a \boxed{\prec} b$ means that for all natural numbers k, the k^{th} instant of a occurs before the k^{th} instant of b: $\forall k \in \mathbb{N} \setminus \{0\}, a[k] \prec b[k]$. Another example is the *coincidence* relation (denoted $\boxed{=}$) imposes a strong synchronous dependency: $a \boxed{=} b$ means that the k^{th} instant of a must be coincident with the k^{th} instant of b: $\forall k \in \mathbb{N} \setminus \{0\}, a[k] \equiv b[k]$. The same mechanism applies for all relations. Informally, the *exclusion* relation (denoted $\boxed{\#}$)

between two clocks a and b specifies that no instants of the clock a coincide with one of the clock b. The *alternatesWith* relation (denoted $\boxed{\sim}$) between two clocks a and b specifies that instants of the clock b are interleaving instants of the clock a.

Expressions are directly defined on clocks. The clock expression *FilteredBy* (denoted $\blacktriangledown$) builds a subclock, *i.e.,* a clock such that all its instants coincide with exactly one instant of the super clock. We use infinite binary words to select those instants of the super clock with which the subclock is coincident. For instance, $a \blacktriangledown 00(01)^\omega$ builds a subclock b of a, such that $\forall k \in \mathbb{N} \setminus \{0\}, b[k] \equiv a[2 * k + 2]$. In this simple example, there is only one 1 in the periodic part of the filter (*i.e.,* (01)), therefore we have a periodic pattern: b is periodic on a with a period of 2 and an offset of 3. When the periodic part only contains 1, then it becomes equivalent to the operator *delayedFor* (denoted $\$_a$). $a \$_a 2$ is equivalent to $a \blacktriangledown 00(1)^\omega$.

A CCSL specification is the conjunction of constraints. This language is central to our approach and is used in Section 6.2, which describes how to express UML interaction semantics with CCSL.

5 Motivating example

This example has been taken from [17] and slightly adapted to our work. We have removed probabilities since they cannot be expressed with CCSL. The example describes a factory with three machines that assemble widgets, and a robot that moves the widgets on demand. Widgets are assembled from two parts, an A part and a B part. Machine 1 processes A parts while machine 2 processes B parts. Machine 3, in turn, assemblies one A part and one B part to make one widget. The robot transports parts from a conveyor belt to the appropriate machine. It is also in charge of moving completed A parts from machine 1 to machine 3, and completed B parts from machine 2 to machine 3. There are always A and B parts available from the conveyor belt. Loading of A parts or B parts can be executed in any order. There are always A and B parts available from the conveyor belt. Loading parts from the conveyor belt, or moving them to machine 3 takes negligible time. The Worst-Case Execution Time (WCET) of processing A parts at machine 1 is 125 seconds, while the WCET of processing B parts at machine 2 is 200 seconds.

6 Going from CCSL partially-ordered traces to UML interaction diagrams

As indicated in Figure 3 the scope of our work is to derive UML sequence and timing diagrams from static and dynamic specifications and partial orderings by means of mappings. Below we describe the elements of our approach being illustrated with the widget factory example.

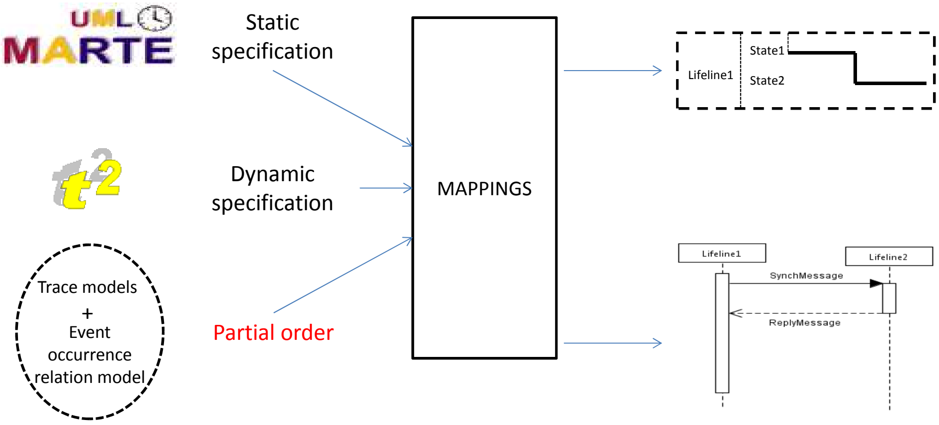

Figure 3: Overview of approach that derives UML interaction diagrams from CCSL partially-ordered traces

6.1 Class and composite structure diagrams as a static specification

In particular, for this report, we use UML class and composite structure diagrams as the static specification. Whereas the class diagram depicts classes, attributes and operations, the composite structure diagram depicts the major components of the system and the relationships between them (*e.g.*, communication relationships). Figure 4 and Figure 5 show, respectively, the class and composite structure diagram for the widget factory. The class diagram basically contains a class for each participant of our system. In the same fashion, the composite structure diagram has four components: the tree machines and the robot. Ports have been annotated with the «ClientServerPort» stereotype which supports a request/reply communication paradigm. To build up the composite diagram, we mainly use the constructs defined in two chapters of the MARTE specification [13]. The Hardware Resource Modeling (HRM) chapter to describe the resources (*i.e.,* processing units) and their properties, and the Time chapter to identify the clocks and apply the constraints.

6.2 CCSL as a dynamic specification

To build the CCSL specification that describes the dynamic behavior of a system, the first step is to declare a clock for each event involved in a system, that is, $MessageSend$, $MessageReceive$, $InternalEventStart$, $InternalEventFinish$, *ReplySend*, *ReplyReceive*. The second step is to define relationships between the clocks following the patterns summarized in Table. 1. Below we illustrate the two steps using the motivating example.

Because our example is provided with class and composite structure diagrams, declared clocks should be associated to class operations and ports. In addition, depending on the

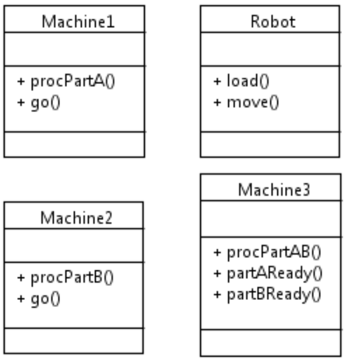

Figure 4: Class diagram for the widget factory

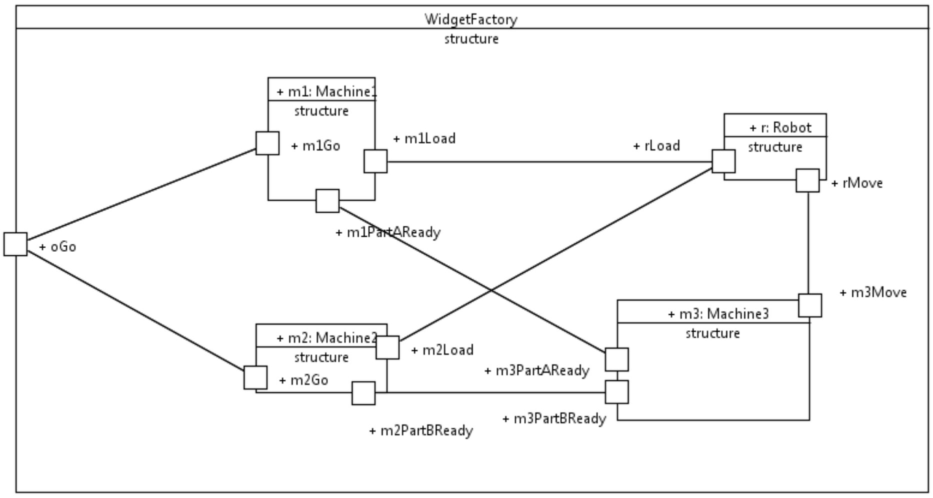

Figure 5: Composite diagram for the widget factory

Kind of time relation	**Kind of involved events**
Precedence	$MessageSend \boxed{\prec} MessageReceive$ $ReplySend \boxed{\prec} ReplyReceive$ $InternalEventStart \boxed{\prec} InternalEventFinish$
Coincidence	$InternalEventStart \boxed{=} MessageSend$ $InternalEventStart \boxed{=} ReplySend$ $InternalEventFinish \boxed{=} MessageSend$ $InternalEventFinish \boxed{=} ReplySend$ $MessageReceive \boxed{=} InternalEventStart$ $ReplyReceive \boxed{=} InternalEventStart$ $MessageReceive \boxed{=} InternalEventFinish$ $ReplyReceive \boxed{=} InternalEventFinish$
Alternance	$MessageSend \boxed{\sim} ReplyReceive$ $MessageReceive \boxed{\sim} ReplySend$

Table 1: Time relations needed for representation of UML interaction diagram semantics

kind of event, the clocks are tagged with one of the following CCSL keywords: *send*, *receive*, *start* or *finish*.

For instance, clock *recvRqLoadA* is associated with port *rLoad* of the robot component and tagged with the keyword *receive*. The ticks of this clock indicate that the request of loading an A part from the conveyor belt has been received by the robot. Clock *sendRpLoadA* is also associated with port *rLoad* but tagged with the keyword *send*. The ticks of this clock suggest that a reply signal has been sent to Machine 1 to inform it about the loading of an A part. Composite structure diagrams mainly specify how components are interconnected to each other to achieve some common objectives [4]. Information about what happens inside a particular component is out of the scope of this kind of diagram. To overcome this limitation, we associate internal clocks (such as $startProcPartA$, $finProcPartA$) with operation $procPartA$.

Once declared, clocks are related each other by means of time constraints. Some constraints relevant for the motivating example follow [3]:

1. $sendRqLoadA \boxed{\prec} recvRqLoadA$
2. $sendRpLoadA \boxed{\prec} recvRpLoadA$
3. $startProcPartA \boxed{\prec} finProcPartA$
4. $recvRqGoA \boxed{=} startGoA$
5. $startGoA \boxed{=} sendRqLoadA$

[3] So far our example illustrates most of kinds of relations except by the coincidence relationships in lines 2 and 7 of Table. 1.

6. $recvRpLoadA \boxed{=} startProcPartA$

7. $finPartAReady * finPartBReady \boxed{=} sendRqProcPartAB$

8. $finPartAReady \boxed{=} sendRpPartAReady$

9. $recvRpPartAReady \boxed{=} finGoA$

10. $sendRqLoadA \boxed{\sim} recvRpLoadA$

11. $recvRqLoadA \boxed{\sim} sendRpLoadA$

12. $recvRqLoadA \boxed{\#} recvRqLoadB$

13. $ResponseTime\ (startProcPartA, finProcPartA, basetime, 0, 125)$

Eq. 1 and Eq. 2 are the classical precedence relations induced by a request/reply communication. The reception (*recvRqLoadA*) always occurs if an emission (*sendRqLoadA*) has occurred.

Eq. 3 is a precedence relation established between the starting and ending point of the internal event *procPartA*.

Next six equations are coincidence relations. Each of them indicates that every time a clock operates, this signal is transferred to its correlated clock and this is done instantaneously. The duration of the sensing action is neglected. Below we explain their functionality with regarding to the example:

- As soon as a message invoking the *go* operation is received by Machine 1, the operation *go* starts, Eq. 4.

- As soon as the internal event *go* starts up in Machine 1, a message is sent to the Robot in order to load an A part, Eq. 5.

- When Machine 1 is notified about the end of the loading, it starts the process of the available A part, Eq. 6.

- At the moment that Machine 3 stops the execution of both behaviors, *partAReady* and *partBReady*, it sends a request to itself in order to assemble A and B parts, Eq. 7.

- In addition, when Machine 3 stops the behavior *partAReady*, a reply message is sent to Machine 1, Eq. 8.

- At the moment that Machine 1 receives the reply of *partAReady*, the execution of *go* is over, Eq. 9.

Eq. 10 and Eq. 11 are alternance relations that help us to control the dispatch of requests or replays. For instance, Eq. 10 makes it possible: a message *i+1* invoking the *load* operation can be sent only if the reply corresponding to the message *i* of the *load* operation has been already received.

Eq. 12 declares an exclusion relation between *sendRqLoadA* and *sendRqLoadB*, *i.e.,* none of their ticks are coincident. The constraint avoids sending, to the robot, two simultaneous orders of loading parts from the conveyor belt.

As a final point, Eq. 13 models the WCET of processing A parts: the time to produce an A part is higher than 0 and lower than 125 ticks.

6.3 Partial order

In our approach, a partial order is represented by (at least) a trace model along with an event occurrence relation model. While a trace model reports the occurrence of clocks (*i.e.,* events), an event occurrence relation model defines temporal and causal relationships between the occurrences of trace models. The relationships indicate that, for some event occurrences, there exist other occurrences that precede (*happen before*) or coincide with (*are simultaneous*) the former occurrences.

In our approach, trace models come from TimeSquare [18] simulations or have been extracted from OTF execution traces which, in turn, are obtained by instrumenting the code of a system. We have built bridges that allows the derivation of UML interaction diagrams from such simulation or execution traces. Thus, one can perform verification and debugging at model level in early (*i.e.,* specification) or advanced (*i.e.,* implementation, deployment) stages of software development. Below, we present the models of what a partial order consists of.

6.3.1 Trace metamodel

A trace model conforms to the metamodel depicted in Figure 6. The principal concepts are as follows. A *Trace* is a sequence of *LogicalStep*s. Each step contains several (simultaneous) *EventOccurrence*s. An event occurrence has an attribute that indicates its state. Among the possible values of this attribute one finds *tick* and *noTick*. Event occurrences can be also marked with the boolean *isClockDead*. A *Reference* associates an event occurrence with a clock or clock expression/relation established in the UML model. When all logical steps refer to a unique *PhysicalBase* of time (*e.g.,* milliseconds), the trace is totally ordered.

6.3.2 Event occurrence relation model

This model defines temporal and causal relationships between occurrences of trace models. The relationships indicate that, for some event occurrences in a specific trace model, there exist other occurrences (in another trace model), that precede (*happen before*) or coincide with (*are simultaneous*) the former occurrences. Unlike a reconciliation specification, that

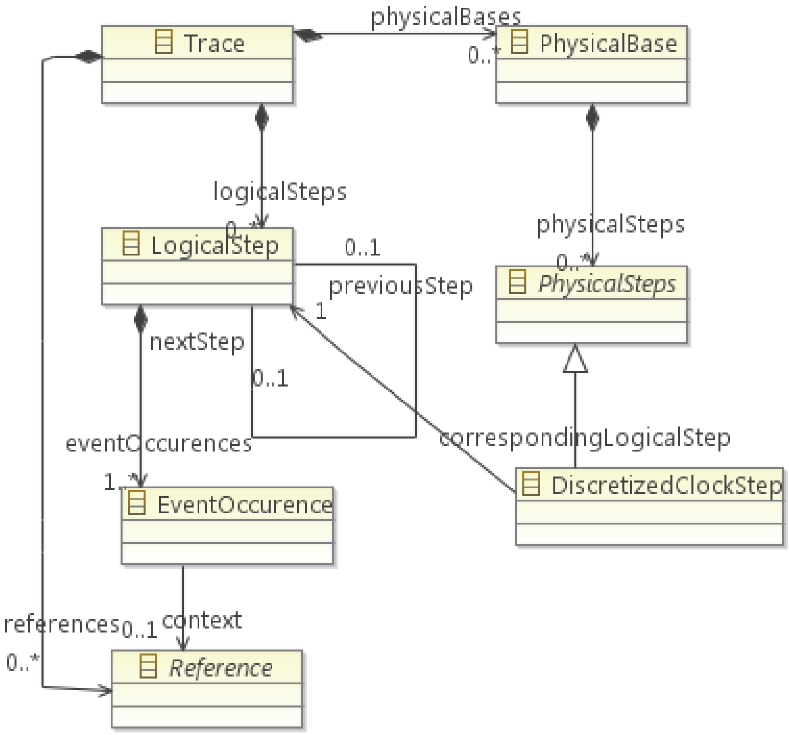

Figure 6: Simplified trace metamodel

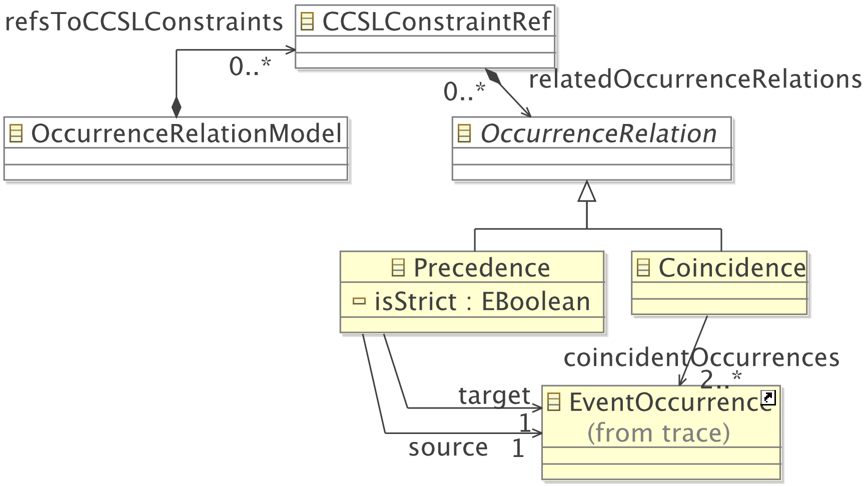

Figure 7: The occurrence relation metamodel

defines relationships between events (*i.e.,* clocks), an occurrence relation model defines relationships between event occurrences (*i.e.,* instants). An occurrence relation model along with a set of trace models represents a partial order. Occurrence relation models conform to a metamodel whose main concepts are displayed in Figure 7. The next paragraph explains the meaning of such concepts.

The root of the metamodel presented in Figure 7 is the entity *OccurrenceRelationModel*. It contains a set of *CCSLConstraintRef*. This kind of elements references the clock constraints of a CCSL specification. It is just a way of sorting the *OccurrenceRelations* with respect to the clock constraints that enforce them. An *OccurrenceRelation* is an abstract concept that represents the possible relationships between occurrences. It is materialized by two kinds of relations, the *Precedence* occurrence relation, which loosely synchronizes two event occurrences and the *Coincidence* event occurrence relation, which forces two event occurrences to be simultaneous. Note that when a *Precedence* is said to be non strict (*isStrict* boolean to false), it covers the union of the *Precedence* and *Coincidence*.

When software components are allocated in different computational units (with their own clock domains which are likely to be unsynchronized), the isolation of execution traces could be desirable to reduce overhead. As a consequence, it is necessary to perform a reconciliation process [3] whose output is a partial order that give us a global vision of the system execution independently of the number of traces. This case is illustrated in Figure 8 where it is assumed that Machine 1 and Robot produce separated traces. As indicated by the legend, the different shapes represent event occurrences of traces of Machine 1 and Robot. In particular, the triangles and squares represent occurrences of the emission and reception of

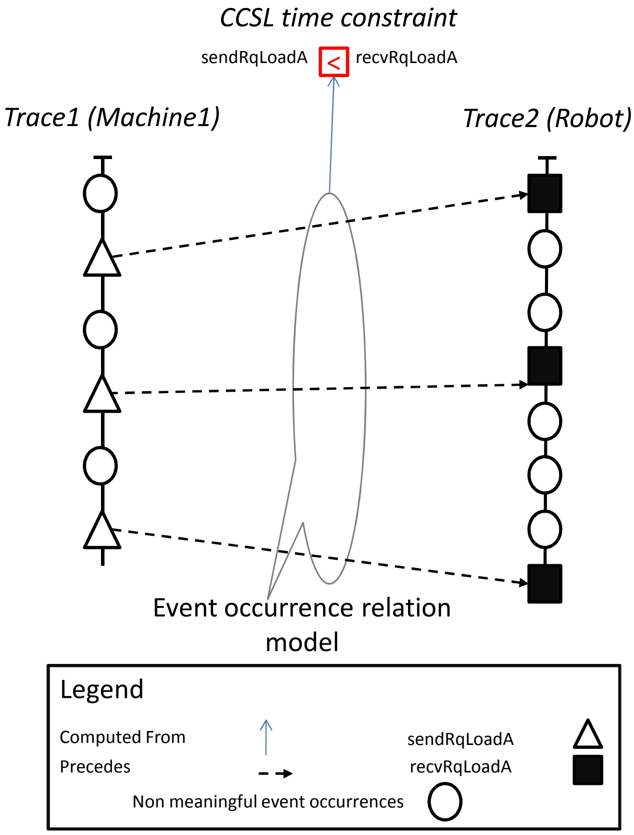

Figure 8: Partial order for the widget factory when traces are isolated

the *load* operation (there referred to as *sendRqLoadA* and *recvRqLoadA*, correspondingly). The dotted arrows, in turn, describe precedence relations between the event occurrences. The two traces plus the occurrences relations constitute the partial order.

Remark on our proof of concept uses only one trace obtained from a simulation. The previous paragraph evokes two execution traces since it worth mentioning that our approach covers different cases.

6.4 Mappings between concepts

This section describes what semantics of UML interaction diagrams can be inferred from our three inputs, that is, static and dynamic specifications and partial order. Table. 2 and Table. 3 show mappings between sequence/timing diagram concepts and concepts of

Sequence diagram concept	**Partial order concept**
Lifeline	Class owner of the Operation or Port pointed by a set of EventOccurrences
Message	Precedence relationships with the form: $MessageSend \prec MessageReceive$ $ReplySend \prec ReplyReceive$
Execution specification	Precedence relationships with the form: $InternalEventStart \prec InternalEventFinish$
Destruction	EventOccurrence with the flag isClockDead equals true
Parallel combined fragment	EventOccurrence scheduling
Coregion	X
Other combined fragments (ALT, LOOP, etc.)	X

Table 2: Mappings between partial orderings (along with static and dynamic specifications) and sequence diagrams

Timing diagram concept	**Partial order concept**
Lifeline	Class owner of the Operation or Port pointed by a set of EventOccurrences
Message	Precedence relationships with the form: $MessageSend \prec MessageReceive$ $ReplySend \prec ReplyReceive$ $InternalEventStart \prec InternalEventFinish$ TimeSquare timing diagrams also require the coincidence relationships
Destruction	EventOccurrences with the flag isClockDead equals true
State or value lifeline	states (*i.e.*, *noTick*, *tick*) or values (*i.e.*, 0, 1) indicated by EventOccurrences

Table 3: Mappings between partial orderings (along with static and dynamic specifications) and timing diagrams

trace and event occurrence relation models. Mappings for messages and parallel combined fragments work under certain conditions (as explained below).

6.4.1 Message

Messages can be classified in two groups *complete* and *incomplete*. The semantics of a *complete* message is that both sender and receiver are known. Synchronous, asynchronous and reply messages have such *complete* semantics. The semantics of an *incomplete* message is that either sender or receiver is known. This is the case for lost and found messages. In this work, we focus on complete messages which can be straightforwardly derived from precedence relationships. To distinguish reply messages from the other kinds of complete messages, we use the following conditions:

- A precedence relationship, $MessageSend \prec MessageReceive$, represents a reply if $MessageReceive$ appears as the first parameter of a coincidence relationship that, in addition, has as a second parameter an event $InternalEventFinish$.

- A precedence relationship, $MessageSend \prec MessageReceive$, represents either a synchronous or asynchronous message if $MessageReceive$ appears as the first parameter of a coincidence relationship that, in addition, has as a second parameter an event $InternalEventStart$.

- Now a message invoking the operation o_2 is classified as synchronous if $ReplyReceive(o_2) = MessageSend(o_2) \; \$ \; 1 \; InternalEventFinish(o_1)$ where o_1 is the last operation executed in $l1$. The lifeline $l1$ is the one invoking the operation o_2 provided by other participant. If the message does not satisfy this constraint is categorized as asynchronous.

6.4.2 Parallel combined fragment

Parallel combined fragment is used when creating a sequence diagram that shows concurrent threads. Fig. 9 gives an example, the bounding box tagged with the keyword PAR delineates the scope of the parallel combined fragment. Horizontal dotted lines delineate the different threads, in this case, two threads. Events within a particular thread are ordered in the usual way. For example, the order of events of lifeline $l1$ with respect to the first thread should be $MessageSend(o_1)$, $ReplyReceive(o_1)$, $MessageReceive(o_2)$, $InternalEventStart(o_2)$, $InternalEventFinish(o_2)$, $ReplaySend(o_2)$. Note that these events follow a pre-established order:

- A message send (*e.g.,* $MessageSend(o_1)$) is ordered immediately before a reply receive (*e.g.,* $ReplyReceive(o_1)$).

- An event start (*e.g.,* $InternalEventStart(o_2)$) is ordered immediately before an event finish (*e.g.,* $InternalEventFinish(o_2)$).

- The events $InternalEventStart(o_2)$, $InternalEventFinish(o_2)$ are in between $MessageReceive(o_2)$, $ReplaySend(o_2)$.

Since the first and second threads are working in parallel, events from the first thread are not causally ordered with respect to events from the second thread. As a result, the order of events of lifeline $l1$ varies from the above mentioned order, that is, the events $MessageSend(o_3)$ and $ReplayReceive(o_3)$ would appear in between.

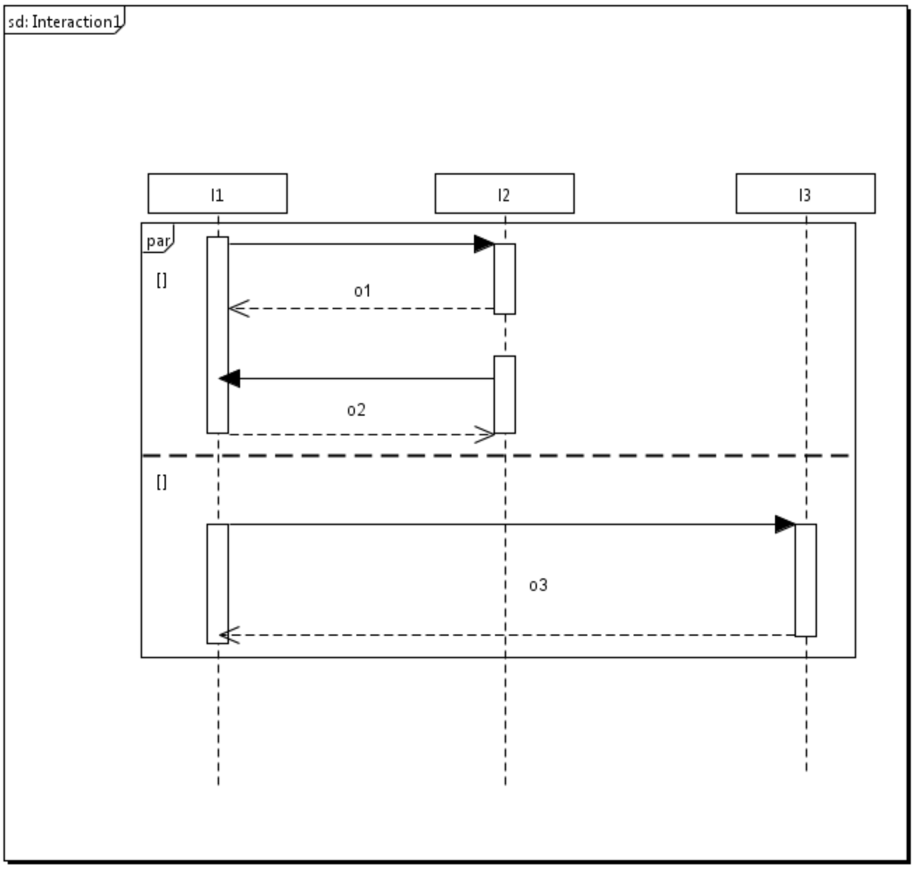

Figure 9: An example of parallel combined fragment

We have designed an algorithm (see Algorithm 1) that detects variations in the order of event occurrences reported in a trace. The assumption is that variations in the pre-established order describe a parallel combined fragment. The algorithm traverses the trace of a given lifeline. It evaluates the type of each event logged in the trace. An event whose type is $MessageSend$, $InternalEventStart$ or $MessageReceive$ is further evaluated in order to see if the event being immediately (or three steps) after it has the proper type. If it is not the case, the event is appended to a list which is, later on, used to graphically represent parallel combined fragments in a sequence diagram. This algorithm targets parallel combined fragments whose messages are synchronous since this kind of message implies events ordered in a deterministic way. Further investigation is needed for covering the inclusion of asynchronous messages in parallel combined fragments.

Algorithm 1 Parallel combined fragment detection

```
DetectParallelCombinedFragment(lifelineTrace)
$fragment \leftarrow \emptyset$
for all $event \in lifelineTrace$ do
  if $event.type \equiv MessageSend$ then
    $event' \leftarrow lifelineTrace.at(lifelineTrace.indexOf(event) + 1)$
    if $event'.type \equiv ReplyReceive$ then
      $Continue$
    end if
  else
    if $event.type \equiv InternalEventStart$ then
      $event' \leftarrow lifelineTrace.at(lifelineTrace.indexOf(event) + 1)$
      if $event'.type \equiv InternalEventStop$ then
        $Continue$
      end if
    else
      if $event.type \equiv MessageReceive$ then
        $event' \leftarrow lifelineTrace.at(lifelineTrace.indexOf(event) + 3)$
        if $event'.type \equiv ReplySend$ then
          $Continue$
        end if
      else
        $Continue$
      end if
    end if
  end if
  $fragment \leftarrow fragment.append(event)$
end for
```

6.4.3 Differences between TimeSquare and UML timing diagram notation

Whereas there are several commercial and open-source tools that allows working with UML sequence diagrams, we only find a few commercial tools supporting UML timing diagrams [19][20]. We have chosen the open-source tools Papyrus [21] and TimeSquare [18] to visualize, respectively, the sequence and timing diagrams resulting from our mappings. Whereas Papyrus graphical representation is mostly aligned with the UML sequence diagram specification, TimeSquare slightly differs from the original UML timing diagram notation. We find the following differences between notations: the compartments of TimeSquare timing diagrams are devoted to events (*i.e.,* clocks) but not to lifelines as it is done in UML. Clocks are represented by pulse trains. Labels for the different states of an event are missing, however one can easily infer them from the graphical representation: when the pulse train has an amplitude equal to one the described state is *tick*, in turn, the state is *noTick* when the amplitude is zero. The arrow styles for messages vary in TimeSquare, dashed arrows represent precedences and filled vertical arrows coincidences. Precedence arrows may represent the semantics of synchronous and replay messages. Although there are variants between the notations, TimeSquare remains nonetheless useful to provide a proof of concept for our approach (see Section 6.6).

6.5 Clock trees

Selection of a kind-of interaction diagram highly depends on the purpose that an embedded software engineer has within. In some cases, both sequence and timing diagrams are required. However, if there is a matter of choosing one of them, we believe that sequence diagrams appear to be a more straightforward representation than timing diagrams when the system has a synchronous nature. It is the other way around for a system whose nature is asynchronous.

Whereas some people would choose the more suitable interaction diagram by reading the system requirements, we use clock trees as an automatic means for that. A clock tree is derived from a CCSL specification. Nodes and links of a clock tree correspond, respectively, to clocks and relations declared in such a CCSL specification. By clock, we mean a terminal clock, an expression, or a list of coincident clocks. In turn, a link labeled with *sub*, $\boxed{\preccurlyeq}$, $\boxed{\prec}$, $\boxed{\#}$describes relations of sub-clocking, precedence, causality, or exclusion. A CCSL specification including several bases of time corresponds to a forest, *i.e.,* a disjoint union of clock trees where the root of each tree is a particular base of time. Now to know what is the system nature, we look at the clock tree: if the tree mostly contains precedence or causality links, then the system has a synchronous nature. Otherwise, if the tree consists of sub-clocking links, then the system is asynchronous.

Figure 10 illustrates an excerpt of the clock tree derived from the CCSL specification of our motivating example. Except by IC113 and IC112, which represents a time response expression, all the nodes are terminal clocks. In addition, the number of precedence links shows that the system has a synchronous nature, therefore, a sequence diagram appears to be the most suitable to understand the system behavior at runtime.

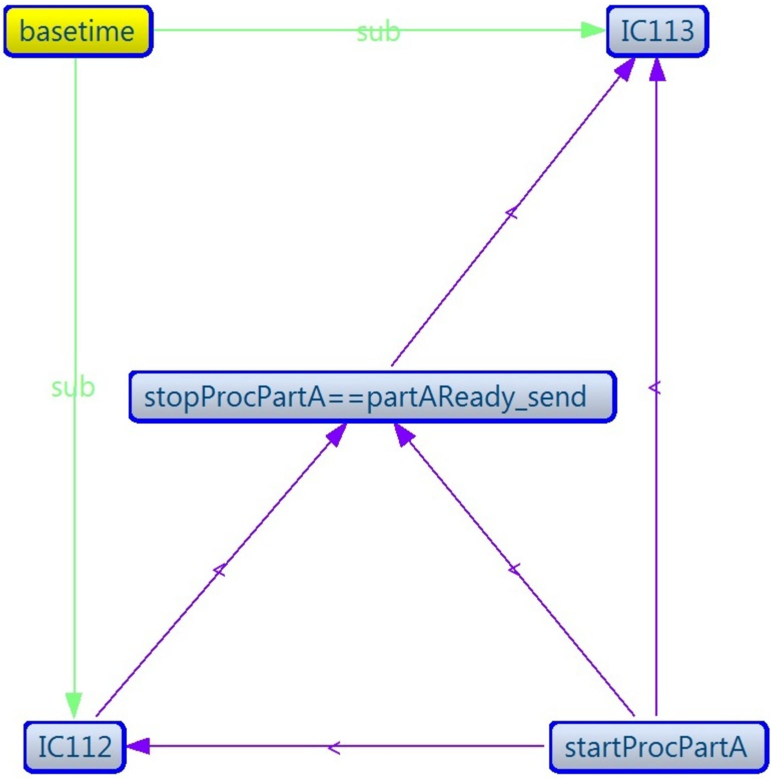

Figure 10: Excerpt of the clock tree derived from the CCSL specification of the widget factory

6.6 Proof of concept

Our proof of concept uses a partial order (one trace and its corresponding event occurrence relation model) obtained from the simulation of the CCSL widget factory specification on the TimeSquare environment. Below we describe in practice how to go from such a partial order toward sequence and timing diagrams.

6.6.1 Derivation of sequence diagrams

It has been implemented in two steps: firstly, the execution of an ATL transformation that takes as input three elements (*i.e.,* a UML model, a CCSL specification and a partial ordering) and gives as output a UML model. This UML model is a copy of the initial UML model plus a new sequence diagram that represents the system at runtime. Secondly, the use of Papyrus tool [21] in order to obtain a graphical visualization for the new sequence diagram. So far, it has been done in a manual fashion, one has to drag and drop the new sequence diagram elements from the Papyrus model explorer view to the editor view. This is a tedious and error-prone task which could be replaced by an automatic process in the future, as it has been already done by Papyrus contributors for class and composite structure diagrams.

Figure 11 shows an excerpt of the sequence diagram derived from a partial ordering which is produced by TimeSquare at executing the widget factory CCSL specification. The screenshot depicts that message exchange happens as expected: machine 1 instructs the robot to give it an A part from the conveyor belt using the load operation. Next machine 1 informs machine 3 the fact that an A part is ready. A reaction to this is to request the robot to move the completed part to machine 3. Remark that the diagram lacks a message indicating the execution of *procPartA* on Machine 1. The reason behind that is a technical problem when the *procPartA* message is dragged from the Papyrus model explorer.

6.6.2 Derivation of timing diagrams

We take benefits from the VCD editor available in TimeSquare to display the timing diagram resulting from the simulation (see Figure 12). One finds the time ruler at the top and the clocks at the left side. The dashed arrows are occurrences of the precedence relation $sendRqLoadA \prec recvRqLoadA$ and the alternance relation $recvRqLoadA \sim sendRpLoadA$. The red small arrows, in turn, are occurrences of the coincidence relation $recvRqLoadA = startLoadA$. Looking at the whole timing diagram shows that the frequency of dispatching *load* messages from Machine 1 is higher that the frequency of Machine 2. This impacts performance of the system. A way of improvement is to add a buffer between Machine 1 and Machine 3.

6.6.3 Results

Hereby a non exhaustive list of results obtained from our proof of concept:

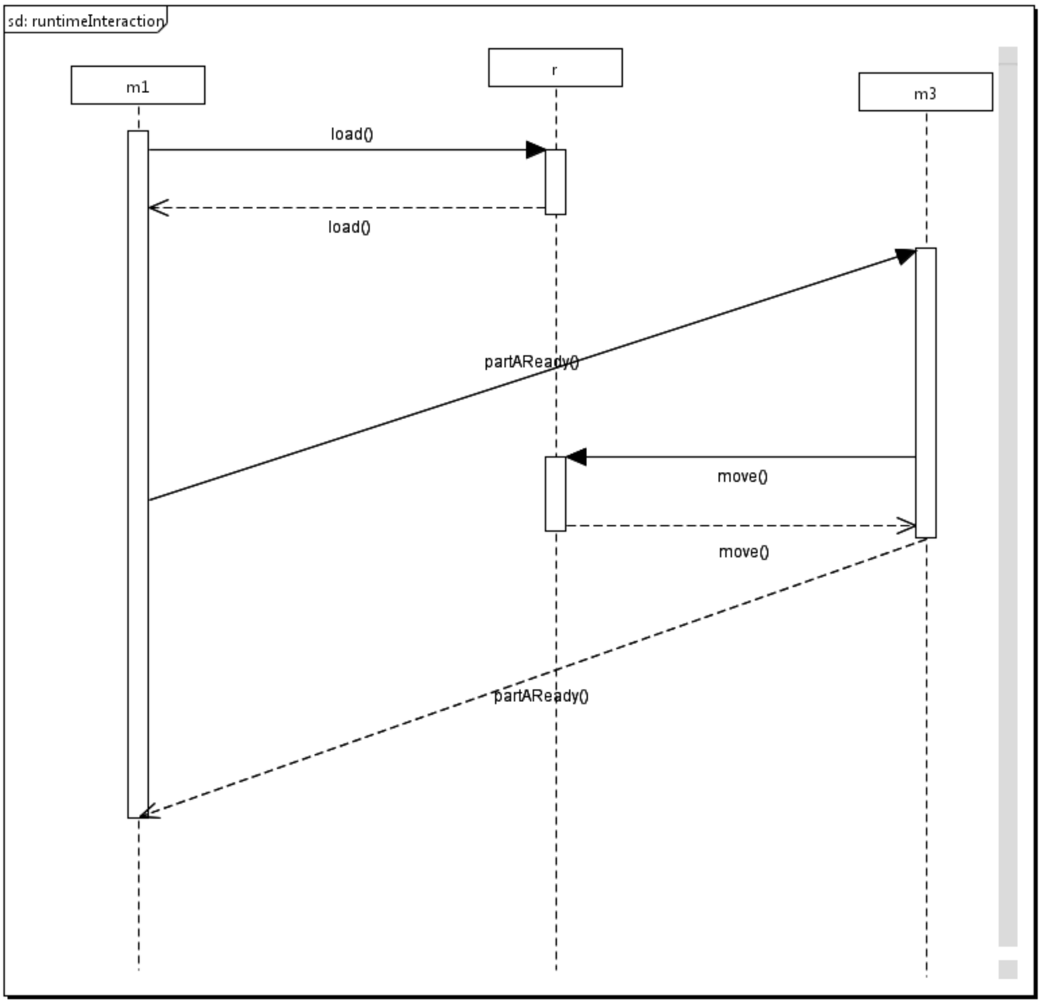

Figure 11: Sequence diagram for the widget factory derived from traces

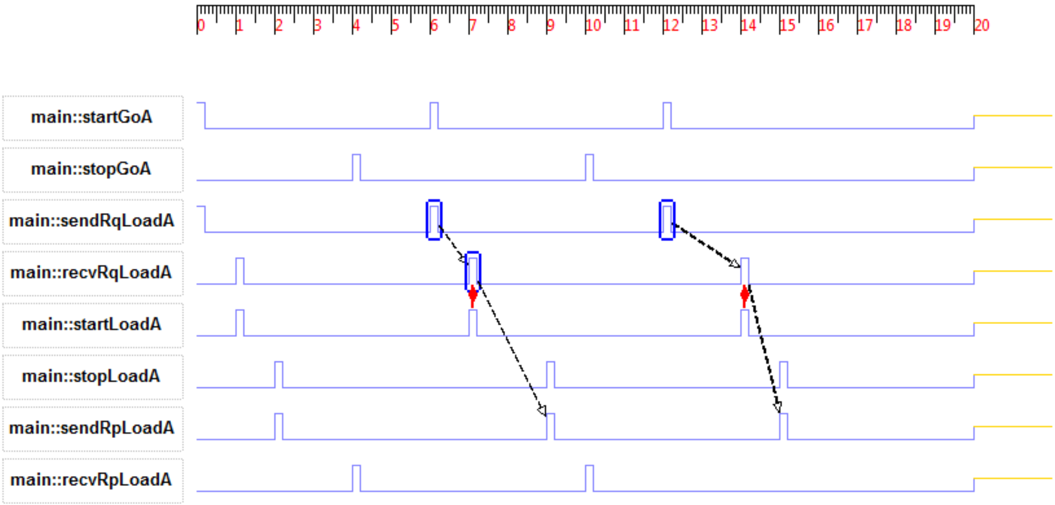

Figure 12: Timing diagram for the widget factory derived from traces

- As indicated in Section 6.4, most of sequence and timing diagram semantics can be inferred from partial orderings and the static and dynamic specifications we have chosen. Only two out of seven sequence diagram semantics cannot be inferred since CCSL semantics (mainly devoted to time modeling) are not expressive enough.
- TimeSquare timing diagrams derivation requires more event occurrence relations than UML sequence diagrams one. That is, TimeSquare timing diagram needs for precedences and coincidences and sequence diagram only requires precedences.
- We have spelled out technical problems at obtaining sequence diagram visualization in Papyrus editor. Although such a technical limitation exists, we demonstrates the feasibility of our approach which opens new perspectives for engineering work. One can imagine the improvement of the prototype and its integration to TimeSquare as a feature that allows animation of UML interaction diagrams derived from traces.
- Although our proof of concept only uses one trace obtained from a simulation, our approach does support the case of multi-traces obtained from executions on a target platform as it is illustrated in [3]. As explained there, the key is the use of a partial order as a pivot and a reconciliation process.

7 Conclusion

The adoption of MDE practices in embedded software community has opened new challenges, among them, the option of verifying and debugging systems at model level. This report is a contribution to this purpose: it proposes partial order as a pivot from which one goes toward UML interaction diagrams (*i.e.,* sequence and/or timing diagrams) in a straightforward way. A partial order consists of (at least one) traces that report the occurrences of events of a system plus the temporal and causal relations between such occurrences. A partial order unfolds the relations defined in CCSL which acts as a dynamic specification of the system behavior. Clocks defined in the CCSL specification point to elements of class and composite structure diagrams (which act as a static specification). Traces come from simulations or executions of the system in a target platform and its number can vary (from 1 to n). Even if the selection of a kind-of interaction diagram highly depends on the user, this report suggests clock trees as a means to select the most appropriate interaction diagram taking into account the system nature. Sequence diagrams appear to be a more suitable representation than timing diagrams when the system has a synchronous nature. It is the other way around for a system whose nature is asynchronous.

A proof of concept demonstrates the applicability of our approach. It shows that CCSL is expressive enough to represent most of the semantics of UML sequence and timing diagrams. Some guidelines for future work are: 1) Solving the technical issues due to the use of Papyrus as editor for displaying the derived sequence diagrams, 2) Animation of derived sequence diagrams, 3) Extension of the approach to support activity diagram derivation and 4) Representation of UML interaction diagram semantics (*e.g.,* loop combined fragment) to which CCSL is currently not expressive enough.

References

[1] Barringer, H., Groce, A., Havelund, K., Smith, M.: Formal analysis of log files. Journal of aerospace computing, information, and communication **7**(11) (2010) 365–390

[2] André, C.: Syntax and semantics of the Clock Constraint Specification Language. Technical Report 6925, INRIA (2009)

[3] Garcés, K., Deantoni, J., Mallet, F.: A model-based approach for reconciliation of polychronous execution traces. In: 37th EUROMICRO Conference on Software Engineering and Advanced Applications. (29 August 2011) 259–266

[4] OMG: Unified Modeling Language, Superstructure. (January 2011) Version 2.4.

[5] Visual paradigm UML tool: UML case tool for software development. (September 2011) http://www.visual-paradigm.com/product/vpuml/.

[6] Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling with time: Specifying the semantics of multimedia sequence diagrams. Software and System Modeling **3**(3) (2004) 181–193

[7] Hammal, Y.: Branching time semantics for uml 2.0 sequence diagrams. In Najm, E., Pradat-Peyre, J.F., Donzeau-Gouge, V., eds.: Formal Techniques for Networked and Distributed Systems - FORTE 2006. Volume 4229 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 259–274

[8] Michelon, L., Costa, S.A.d., Ribeiro, L.: Formal specification and verification of real-time systems using Graph Grammars. Journal of the Brazilian Computer Society **13** (12 2007) 51 – 68

[9] Iyenghar, P., Westerkamp, C., Wuebbelmann, J., Pulvermueller, E.: A model based approach for debugging embedded systems in real-time. In: Proceedings of the tenth ACM international conference on Embedded software. EMSOFT '10, New York, NY, USA, ACM (2010) 69–78

[10] Iyenghar, P., Westerkamp, C., Wuebbelmann, J., Pulvermueller, E.: Design level debugging of timing behavior in embedded systems: Using a model-based approach. In: 9th IEEE International Conference on Industrial Informatics. (2011) 889–894

[11] Apvrille, L., Becoulet, A.: Fast and multi-platform prototyping of embedded systems from uml/sysml models. In: SAME 14th edition : Sophia Antipolis Microelectronics Forum. (2011)

[12] UPPAAL: UPPAAL Home. http://www.uppaal.org.

[13] OMG: UML Profile for MARTE, v1.0. Object Management Group. (November 2009) formal/2009-11-02.

[14] André, C., Mallet, F., de Simone, R.: Modeling time(s). In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: MoDELS. Volume 4735 of Lecture Notes in Computer Science., Springer (2007) 559–573

[15] OMG: UML Superstructure, v2.2. Object Management Group. (February 2009) formal/2009-02-02.

[16] DeAntoni, J., Mallet, F., André, C.: TimeSquare: on the formal execution of UML and DSL models. Tool session of the 4th Model driven development for distributed real time systems (2008)

[17] Bennett, A., Field, A., Woodside, C.: Experimental evaluation of the uml profile for schedulability, performance, and time. In Baar, T., Strohmeier, A., Moreira, A., Mellor, S., eds.: UML 2004 - The Unified Modeling Language. Modelling Languages and Applications. Volume 3273 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2004) 143–157

[18] AOSTE: TimeSquare Home. http://timesquare.inria.fr/.

[19] ALTOVA: UML Timing diagrams. (December 2011) http://www.altova.com/umodel/timing-diagrams.html.

[20] Metamill: Product features. (December 2011) http://www.metamill.com/features.html.

[21] team, P.: Papyrus UML. (Accessed on Dec. 2011) http://www.papyrusuml.org.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399