TimeSquare: Treat Your Models with Logical Time[⋆]

Julien DeAntoni and Frédéric Mallet

Aoste Team-Project
Université Nice Sophia Antipolis
I3S - UMR CNRS 7271, INRIA Sophia Antipolis Méditerranée
2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex, France
{julien.deantoni,frederic.mallet}@inria.fr
http://www-sop.inria.fr/aoste/

Abstract. TimeSquare is an Eclipse and model-based environment for the specification, analysis and verification of causal and temporal constraints. It implements the MARTE Time Model and its specification language, the Clock Constraint Specification Language (CCSL). Both MARTE and CCSL heavily rely on logical time, made popular by its use in distributed systems and synchronous languages. Logical Time provides a relaxed form of time that is functional, elastic (can be abstracted or refined) and multiform. TimeSquare is based on the latest model-driven technology so that more than 60% of its code is automatically generated. It provides an XText-based editor of constraints, a polychronous clock calculus engine able to process a partial order conforming to the set of constraints and it supports several simulation policies. It has been devised to be connected to several back-ends developed as new plugins to produce timing diagrams, animate UML models, or execute Java code amongst others.

Keywords: Embedded systems, Polychronous specifications, Logical Time, Model-Driven Engineering.

1 Introduction

Models abstract away the irrelevant aspects of a system to focus on what is important for a given purpose. Model-driven engineering provides tools and techniques to deal with models. These models are nowadays mainly structural but can often be refined with a behavioral description. The behavioral description is usually a specific implementation of externally defined behavioral requirements. To fully benefit from models right from the requirements we propose to specify behavioral requirements as logical time constraints directly linked to the model.

⋆ This work has been partially supported by the RT-SIMEX ANR project (http://www.rtsimex.org) and the PRESTO ARTEMIS project (http://www.presto-embedded.eu/).

This is done by using the Time Model from MARTE conjointly with its formal companion language CCSL (Clock Constraint Specification Language [1]). This approach is tooled by the TimeSquare framework, which is a set of Eclipse plugins that implement the model-based declarative language CCSL and provide support for the analysis and execution of CCSL specifications.

In this paper, we overview the main functionality of TimeSquare. Our tool, which is itself based on a model-driven approach, allows for the enrichment of models with formal annotations by using semantic models. CCSL concrete syntax is based on Xtext (`http://www.eclipse.org/Xtext/`) so that the user directly constructs an EMF model while typing. This model can be parsed easily to provide useful information like the clock tree graph that represents the polychronous specification in a graphical way [6]. Keeping the specification as a model enables a better integration with a model-driven approach because the model, the formal language and the solver are in the same technological space. The main benefits is the ability to link specification as well as results directly to the models. The feedback to the user is then greatly improved compared with transformational techniques, which translate a model to an existing formal language. The output of TimeSquare is also an EMF model that defines a specific partial order of events, which represents the trace of the simulation. It is important to notice that a single simulation provides a partial order and consequently captures several possible executions (several total orders). It is possible to subscribe to specific events during the construction of this trace by using the extension point mechanism provided by Eclipse. User-defined or domain-specific backends can be deployed by registering to selected events.

The architecture of TimeSquare is shown in Figure 1. Straight arrows indicate the model flows, whereas dashed arrows represent the links between two models. The trace model is directly linked to CCSL model elements, which in turns are linked to other (EMF) model elements.

The paper organization follows this architecture. Section 2, after a brief overview of CCSL semantics, describes the CCSL concrete syntax and tooling. Section 3 explains how the solver produces the trace model according to the simulation policy. Finally, before concluding, section 3.1 details the back-end mechanism and details some of the main existing back-ends.

2 CCSL Specifications

2.1 Semantics

Contrary to most real-time constraint tools, we use a polychronous time model that allows the duration and time values to be expressed relatively to other clocks, and not only relatively to a common chronometric clock counting physical time. For instance, a duration can be expressed relative to the clock cycle of a given processor core or bus. In many current electronic devices, the clock cycle varies according to the battery level or some other optimization criteria. This kind of time is named logical time and has been used in distributed systems [5,4]

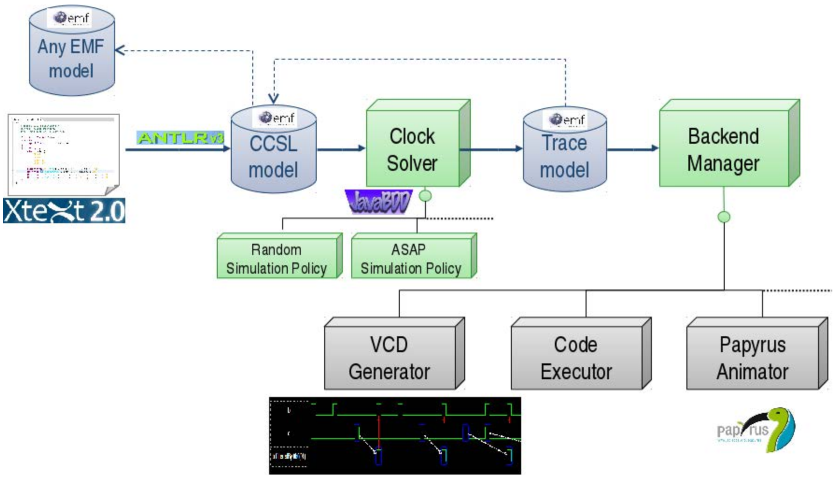


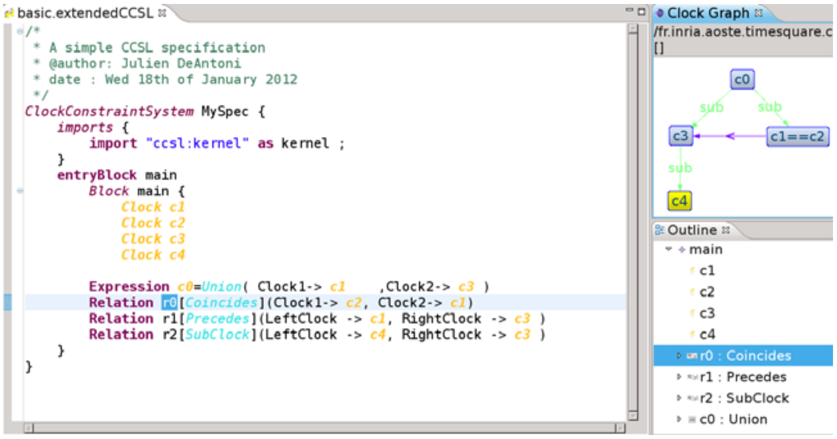
Fig. 1. Big Picture of the TimeSquare Architecture

for its ability to represent (untimed) causalities but also in synchronous language where it has prooved to be meaningful from requirements to implementations [2].

In MARTE, the Time Model relies on logical time. In this context, a *clock* is a totally ordered set of *instants*. A *time structure* is a set of clocks C and a set of relations on instants. I denotes the union of all instants of all clocks within a given time structure. We consider two kinds of relations: *causal* and *temporal* ones. The basic causal relation over I is causality/*dependency*, $i \in I, j \in I, i \preccurlyeq j$ means i causes j or j depends on i, *i.e.,* if j occurs then i also occurs. The three basic temporal relations over I are *precedence* ($\prec$), *coincidence* ($\equiv$), and *exclusion* ($\#$). For any instants i and j in a time structure, $i \prec j$ means that the only acceptable execution traces are those where i occurs strictly before (precedes) j. $i \equiv j$ imposes instants i and j to be coincident, *i.e.,* they must always occur at the same execution step, both or none . $i \# j$ forbids the coincidence of the two instants, *i.e.,* they cannot occur at the same execution step. Note that, some consistency rules must be enforced between causal and temporal relations. $i \preccurlyeq j$ can be refined either as $i \prec j$ or $i \equiv j$, but j can never precede i. Furthermore, we do not assume a global notion of time. Temporality is given by the *precedence* binary relation, which is partial, asymmetric (*i.e.,* antisymmetric and irreflexive) and transitive. The *coincidence* binary relation is an equivalence relation on instants, *i.e.,* reflexive, symmetric and transitive. Specifying a full time structure using only instant relations is not realistic since clocks are usually infinite sets of instants. Thus, CCSL defines a set of relations and expressions between clocks that apply to infinitely many instant relations. Please refer to [1] to learn about CCSL semantics.

2.2 Implementation

The clock constraint specification language (CCSL) complements structural models by formally defining a set of kernel clock constraints, which apply to infinitely many instant relations. The operational semantics of CCSL constraints is defined in a technical report [1]. Some recurrent constraints from a specific domain can be complex. To ease the application of such complex constraints, libraries of user-defined constraints can be built by composing existing constraints. This language and the library mechanism is defined in a metamodel accessible here: `http://timesquare.inria.fr/resources/metamodel`. This metamodel can be instantiated from two different classes depending on whether the user wants to create a CCSL specification or a library. Because using the ecore reflective editor provided by EMF is not suitable for any user, we created a textual concrete syntax using XText. XText automatically generates a textual editor for a given EMF metamodel and allows for customizing the concrete syntax. Then, when using the textual editor, the corresponding EMF model is automatically built. Amongst other things, direct links to external EMF models are supported. In the CCSL editor, we use such links to map CCSL clocks to EMF model elements such as the UML model elements whose execution is triggered by the CCSL clocks. Such direct links are important to help the user in the specification of constraints and the creation of a coherent specification (completion, detection of errors on the fly, tips, etc). Two kinds of model can be imported in a CCSL specification: external libraries and EMF-based models. If a library is imported, the Xtext editor automatically proposes, as a completion mechanism, the relations and the expressions from the library. It also checks the parameters provided and proposes some changes if a problem is detected. Such customization features are very helpful to build the specification. Figure 2 illustrates a simple CCSL specification being edited with the XText constraint editor.



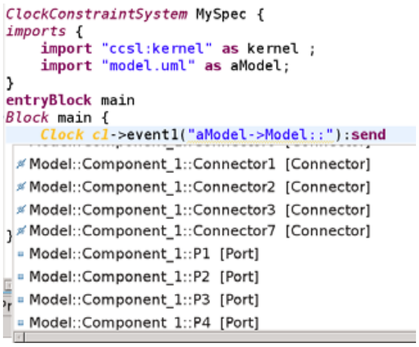
```
/*
 * A simple CCSL specification
 * @author: Julien DeAntoni
 * date : Wed 18th of January 2012
 */
ClockConstraintSystem MySpec {
    imports {
        import "ccsl:kernel" as kernel ;
    }
    entryBlock main
        Block main {
            Clock c1
            Clock c2
            Clock c3
            Clock c4

        Expression c0=Union( Clock1-> c1     ,Clock2-> c3 )
        Relation r0[Coincides](Clock1-> c2, Clock2-> c1)
        Relation r1[Precedes](LeftClock -> c1, RightClock -> c3 )
        Relation r2[SubClock](LeftClock -> c4, RightClock -> c3 )
    }
}
```

Fig. 2. A simple CCSL specification in TimeSquare

If an EMF model is imported in a CCSL specification, all the elements from the model that own a “name” property will be accessible and possibly constrained. Figure 3 shows a part of the previous CCSL specification where an import from a

UML model is done. It allows enriching the *Clock* declaration with the structural element from the UML model (here subject to completion). The meaning of the link can also be specified: *i.e.*,, the clock ticks can represent the starting/finishing of a behavior, the sending/reception of a message...



```
ClockConstraintSystem MySpec {
imports {
    import "ccsl:kernel" as kernel ;
    import "model.uml" as aModel;
}
entryBlock main
Block main {
    Clock c1->event1("aModel->Model::"):send
```

```
Model::Component_1::Connector1  [Connector]
Model::Component_1::Connector2  [Connector]
Model::Component_1::Connector3  [Connector]
Model::Component_1::Connector7  [Connector]
Model::Component_1::P1  [Port]
Model::Component_1::P2  [Port]
Model::Component_1::P3  [Port]
Model::Component_1::P4  [Port]
```

Fig. 3. Link between a model (UML here) and a CCSL specification, helped by completion

3 Simulation

The formal operational semantics of CCSL constraints makes CCSL specifications executable. A *run* of a time system is an infinite sequence of *steps* (if no deadlocks are found by the solver). During a step, a Boolean decision diagram represents the set of acceptable sets of clocks that can tick. If the CCSL specification contains assertion(s), then the Boolean decision diagram also represents the state of the assertion (violated or not). Assertions never change the clocks that can tick. It has been used in the RT-Simex project to check if a specific execution trace is correct with regards to a CCSL specification [3]. If the CCSL specification is deterministic, there exists a single set; if not, a simulation policy is used to choose amongst the possible solutions. TimeSquare offers several simulation policies (Random, As soon as possible, etc). It is possible for a user to add a new simulation policy by using a specific TimeSquare extension point. The choice of the simulation policy, the number of steps to compute as well as the choices about debugging information are integrated in the existing eclipse configuration mechanism so that a run or a debug (step by step) of a CCSL specification is accessible as in other languages like java.

3.1 Analysis Features and Back-Ends

TimeSquare can be used in various model-driven approaches. Depending on the domain, users are interested in different feedback or analysis of the results. To allow an easy integration of TimeSquare in various domains, we implemented a back-end manager, which enables the easy addition of user-defined back-ends.

The back-end manager receives the status of the clock (it ticks or not) at each simulation step. It also receives the status of relations (causality and coincidence) as well as the status of the assertions (violated or not). By using a specific extension point, a developer can create a back-end that subscribes to some of these events. The registered back-end are then notified when the events they subscribed to occur during the simulation step. We present in the remainder of this section the three main backends: the VCD diagram creator, the papyrus animator and the code executor.

VCD Diagram Creator: VCD is a format defined as part of IEEE1364 and is mainly used in the electronic design domain. It is very close to the UML timing diagram and represents the evolution of each event (Clock) *vs.* time evolution, represented horizontally. It classically represents a total order of events. Because TimeSquare provides a trace which is only partially ordered, the classical VCD features have been extended to graphically represent such a partial order. On Figure 4, a simple VCD is represented. It results from the simulation of the CCSL specification represented on Figure 2 where the `c0` clock is hidden to simplify the reading. We can notice the optional presence of two kinds of links between the ticks of the clocks: blue arrows, which represent causalities (loose synchronizations) and red links, which represent coincidences (strong synchronizations). The result is that the partial order is valid as long as the red links are not broken and the blue arrows never go back in time.

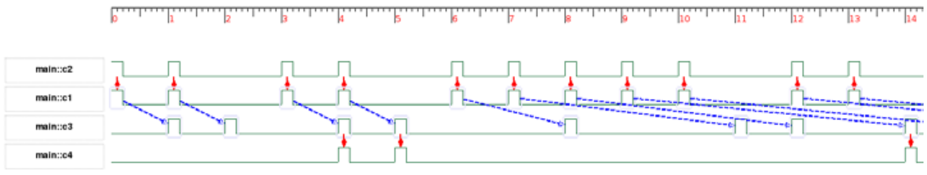
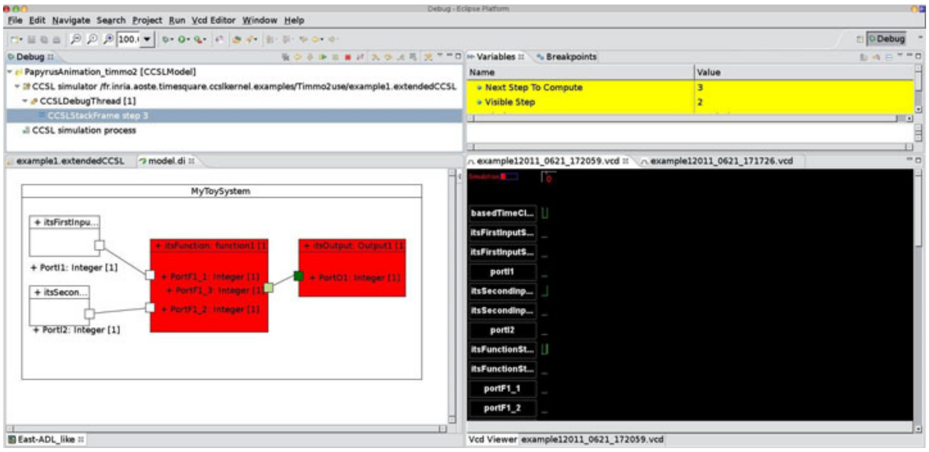


Fig. 4. The extended VCD diagram back-end

Papyrus Diagram Animator: When a CCSL specification is linked to a UML model, the model is often represented graphically in a UML tool. Papyrus (`http://www.eclipse.com/Papyrus`) is an open source UML tool integrated with eclipse EMF and GMF. The papyrus animator provides a graphical animation of the UML diagrams during the simulation. The kind of graphical animation depends on the "meaning" of the event linked to the UML model (send, reveive, start, etc). This animation provides a very convenient feedback to the user who wants to understand what happens in the model according to the constraints he wrote. Additionally to graphical animation, the Papyrus animator adds *comments* to the UML model elements that represent their activation trace, keeping this way a trace of the simulation directly in the UML model. The Papyrus animator is shown conjointly (and synchronized with) the VCD diagram on Figure 5.



Name	Value
Next Step To Compute	3
Visible Step	2

Fig. 5. The animation of a UML model and the associated timing diagram in TimeSquare

Code Executor: When a software is prototyped, it can be convenient to run some piece of code in order to provide application specific feedback. For instance we developped a simple digital filter by using UML composite structure in Papyrus and we added constraints on it representing its synchronizations (so that the diagram can be animated conjointly with the VCD diagram). To test our algorithm and ease the debugging of the synchronization in the model, we used the JAVA code executor. It allows the declaration of object and the launch of specific method of these objects when a desired event occurs (tick of a clock, etc). It can be used, as in the digital filter, to represent the data manipulation of the filter and to graphically represent the internal state of the memory. It can also be used to pop-up information windows when an assertion is violated, etc.

Clock Graph: To allow static analysis as, for instance the one described in [6], TimeSquare is able to build statically a clock graph that depicts the synchronous/asynchronous relations between clocks. This specific mechanism is not a back-end *per se* because it does not depend on the dynamics of the model but it is a very useful feature to deal with polychronous specifications. A simple CCSL specification, the corresponding and synchronized EMF model in the outline and the associated clock graph are represented on Figure 6. The vertices are the clocks and the edges are the clock relationships: *sub* denotes a subclocking and therefore a synchronous relationship, whereas $<$ denotes a precedence by nature asynchronous. When two clocks are synchronous, they are merged into a single vertex (as $c1 == c2$). This graph shows that the specification is fully synchronous: $c0$ is the super clock of both $c1$ and $c3$. $c1$ in turns is a super clock of $c4$ and is synchronous with $c2$. It also shows the precedence relationship between $c1$ and $c3$.

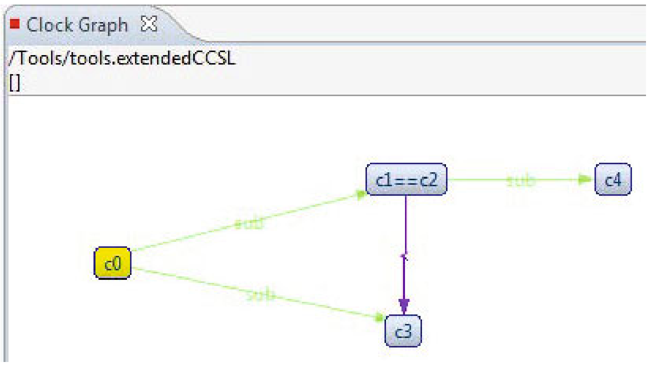


Fig. 6. Clock Graph extracted from a CCSL specification

4 Conclusions

This paper briefly presents TimeSquare. It is a model-based tool well integrated in the Model Driven Development process. Its goal is to ease the use of the formal declarative language CCSL and provides analysis support. Additionally, we wanted to develop it by using model driven technology; in one hand it has helped in the development of our tool and on the other hand it put the tool in the same technological space than the model under development. The main benefit is the direct feedback offered to the users during the simulation. A video demonstration is available from the TimeSquare website (in French): `http://timesquare.inria.fr/`. Finally, while not presented here, it also supports a form of runtime analysis through the generation of VHDL or Esterel observers.

References

1. André, C.: Syntax and semantics of the clock constraint specification language (ccsl). Research Report 6925, INRIA (May 2009)
2. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., De Simone, R.: The synchronous languages twelve years later. Proceedings of the IEEE, 64–83 (2003)
3. Deantoni, J., Mallet, F., Thomas, F., Reydet, G., Babau, J.-P., Mraidha, C., Gauthier, L., Rioux, L., Sordon, N.: RT-simex: retro-analysis of execution traces. In: In, K.J., Sullivan, G.-C. (eds.) SIGSOFT FSE, Santa Fe, États-Unis, pp. 377–378 (2010) ISBN 978-1-60558-791-2
4. Fidge, C.: Logical time in distributed computing systems. Computer 24(8), 28–33 (2002)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
6. Yu, H., Talpin, J.-P., Besnard, L., Gautier, T., Mallet, F., André, C., de Simone, R.: Polychronous analysis of timing constraints in UML MARTE. In: IEEE Int. W. on Model-Based Engineering for Real-Time Embedded Systems Design, Parador of Carmona, Spain, pp. 145–151 (2010)