Analyzing and Simulating Time Descriptions from UML/MARTE CCSL*

Judith Peters
PhD advisor: Rolf Drechsler

Institute of Computer Science, University of Bremen, Bremen 28359, Germany
jpeters@informatik.uni-bremen.de

Abstract. The complexity of modern embedded systems makes it inevitable to consider higher abstraction levels in the design process to overcome problems in acceptable time and effort. In higher abstraction levels, the utilization of functional requirements is quite advanced, while the utilization of non-functional requirements like timing still is an open problem. We aim to address this problem utilizing the timing definitions from UML/MARTE CCSL.

Keywords: CCSL, UML, MARTE, SystemC, Formal Methods

1 Introduction

Modern embedded systems are growing to a huge complexity, making classical design tasks error-prone and time-consuming. As a consequence, novel design flows introduced several abstraction levels to overcome this problem. At this stage, the classical level of highest abstraction is the *Electronic Systems Level* (ESL), where SystemC and other high-level programming languages are used to describe the system. But still, a big gap remains between textual specification and ESL. In the last decade, modeling languages like the *Unified Modeling Language* (UML, [2]) provided a "bridge" between the given specification and its initial implementation [4]. In the design of embedded and cyber-physical systems, particularly the *Modeling and Analysis of Real-time and Embedded systems* profile (MARTE, [1]) finds considerable attention.

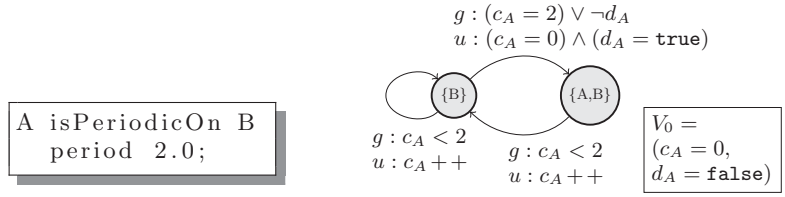
In this field, the utilization of non-functional requirements like timing is still an open problem. MARTE provides a special language for timing specification: the *Clock Constraint Specification Language* (CCSL), which relies on describing clocks and instants. In our project we are going to utilize this language for classical design tasks such as verification or code-generation.

2 Design and More – A Generic Representation of CCSL

To utilize the textual CCSL specification, we introduce a generic automaton representation of it. A lot of approaches have already been proposed to directly

* This work was supported by the Graduate School SyDe, funded by the German Excellence Initiative within the University of Bremen's institutional strategy.

R. Drechsler, U. Kühne (eds.), *Formal Modeling and Verification of Cyber-Physical Systems,*
DOI 10.1007/978-3-658-09994-7_17, © Springer Fachmedien Wiesbaden 2015



(a) CCSL constraints (b) Corresponding automaton

Fig. 1. Generic representation of CCSL constraints

transform the CCSL constraints to input for certain model checkers [5] or to ESL descriptions [3]. We transform it to a more generic representation which can be used for several tasks at the same time.

The main concept of our approach is the *ticking set*, which is the set of clocks $c \in \mathcal{C}$ ticking in one simulation step. All clock behavior can be modeled as a movement between these ticking sets, making the ticking sets *states* and the movement between them the *transitions*. These transitions can be restricted according to the CCSL constraints using guard functions over global variables. These variables are manipulated using update functions.

For a more detailed explanation, consider the CCSL specification from Fig. 1(a). It means that `B` is not restricted and can tick whenever it wants, while `A` is a subclock of `B`. Thus, `A` cannot tick alone but has to tick coincidently with `B`. For the resulting automaton, this gives the ticking sets {`B`} and {`A`, `B`} (see Fig. 1(b)). Furthermore, as the period of `A` is 2.0, we can conclude that the ticking set {`A`, `B`} can never be followed by itself, i. e. this transition is not needed. Finally this leads to two states and three transitions (see Fig. 1(b)).

To enforce the periodicity, now a counter c_A for the period is added and additionally a Boolean variable d_A to represent, if the subclock `A` has already ticked. The initial values V_0 are $c_A = 0$ and $d_A = \texttt{false}$. For every transition leading to a state not containing `A`, c_A is increased, while the other transition resets the counter to zero and sets d_A, representing, that `A` has ticked now and the period starts again (see the conditions u in Fig. 1(b)). The transition to the ticking set including `A` can only be taken if the counter is high enough, while the other transitions can only be taken if it is low enough (see g in Fig. 1(b)). Now, the resulting automaton can be used for various design tasks such as e. g. verification or code-generation.

3 Generating SystemC

As the behavior and thereby the automaton and its analysis time is growing exponentially in the clock number, we developed a faster way to simulate and test the constraints together with SystemC applications [3]. This is an alternative to the automaton approach especially for systems with high numbers of clocks and, thus, long verification times. Our code generation scheme extends a given functional implementation in SystemC with timing. To generate the behavior, we extend the existing implementation by a *TimeController* as shown in Fig. 2.

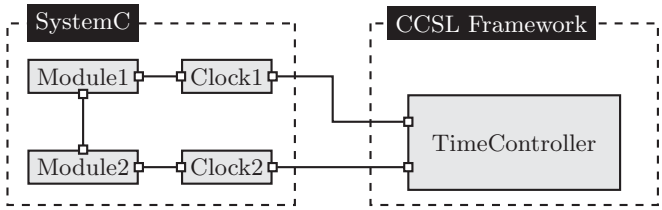


Fig. 2. Example SystemC implementation extended by CCSL Framework

CCSL constraints can be divided in two kinds: combination and future constraints. Combination constraints state, in what combinations clocks may or may not tick, while future constraints define times in the future from a given moment on, where other clocks have to tick or not to tick. To store the future constraint information, we use *ClockMonitor* objects. One object for every clock stores, which constraints it applies to other clocks and what restrictions are applied to itself. The combination constraints are represented by *Bind* objects. Finally, the TimeController uses lists of clocks that *can, cannot* or *must* tick in every step to represent the behavior. These lists are updated according to the constraints, until all clocks are distributed between *must* and *cannot*. The clocks in *must* form finally the ticking set of that simulation step.

4 Conclusion and Future Work

In the recent past, we developed a generic representation of CCSL constraints in terms of automata, which can represent the whole breadth of CCSL constraints. As the analysis of these automata is time-consuming, we developed a simple and fast code-generation scheme, which can be used for fast tests and simulations. Now, we want to combine the two approaches to get better simulations (regarding non-deterministic choices and clock-dependencies) and to improve the automaton evaluation towards bounded model checking and symbolic representation to make its use more feasible.

References

1. Object Management Group: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Object Management Group (2011)
2. Object Management Group: OMG Unified Modeling Language TM (OMG UML) Superstructure. Object Management Group (2011)
3. Peters, J., Wille, R., Drechsler, R.: Generating SystemC Implementations for Clock Constraints Specified in UML/MARTE CCSL. International Conference on Engineering of Complex Computer Systems (ICECCS), 116–125 (2014)
4. Drechsler, R., Soeken, M., Wille, R.: Formal Specification Level: Towards Verification-driven Design Based on Natural Language Processing. Forum on Specification and Design Languages (FDL), 53–58 (2012)
5. Mallet, F., Yin, L.: Correct Transformation from CCSL to Promela for verification. Institut National de Recherche en Informatique et en Automatique, (2012)