Multi-core Code Generation from Polychronous Programs with Time-Predictable Properties

Zhibin Yang, Jean-Paul Bodeveix, and Mamoun Filali

IRIT-CNRS, Université de Toulouse, France
{Zhibin.Yang,bodeveix,filali}@irit.fr

Abstract. Synchronous programming models capture concurrency in computation quite naturally, especially in its dataflow multi-clock (polychronous) flavor. With the rising importance of multi-core processors in safety-critical embedded systems or cyber-physical systems (CPS), there is a growing need for model-driven generation of multi-threaded code for multi-core systems. This paper proposes a build method of time-predictable system on multi-core, based on synchronous-model development. At the modeling level, the synchronous abstraction allows deterministic time semantics. Thus synchronous programming is a good choice for time-predictable system design. At the compiler level, the verified compiler from the synchronous language SIGNAL to our intermediate representation (S-CGA, a variant of guarded actions) and to multi-threaded code, preserves the time predictability. At the platform level, we propose a time-predictable multi-core architecture model in AADL (Architecture Analysis and Design Language), and then we map the multi-threaded code to this model. Therefore, our method integrates time predictability across several design layers.

Keywords: Synchronous languages, SIGNAL, Guarded actions, Verified compiler, Multi-core, Time predictability, AADL

1 Introduction

Safety-critical embedded systems or cyber-physical systems (CPS) distinguish themselves from general purpose computing systems by several characteristics, such as failure to meet deadlines may cause a catastrophic or at least highly undesirable system failure. Time-predictable system design [1, 21, 20] is concerned with the challenge of building systems in such a way that timing requirements can be guaranteed from the design. This means we can predict the system timing statically. With the widespread advent of multi-core processors in this category of systems, it further aggravates the complexity of timing analysis.

The synchronous abstraction allows deterministic time semantics. Therefore synchronous programming is a good choice for time-predictable system design. There are several synchronous languages, such as ESTEREL [5], LUSTRE [12] and QUARTZ [16] based on the *perfect synchrony* paradigm, and SIGNAL [4] based on the *polychrony* paradigm.

An integration infrastructure for different synchronous languages has gained a lot of interests in recent years [6, 19]. A classical solution is to use an intermediate representation. Guarded commands [10], also called *asynchronous guarded actions* by J. Brandt et al. [6], are a well-established concept for the description of concurrent systems. In the spirit of the guarded commands, J. Brandt et al. propose *synchronous guarded actions* [8] as an intermediate representation for their QUARTZ compiler. As the name suggests, it follows the synchronous model. Hence, the behavior (control flow as well as data flow) is basically described by sets of guarded actions of the form $\langle \gamma \Rightarrow \mathcal{A} \rangle$. The boolean condition γ is called the guard and $\mathcal{A}$ is called the action. To support the integration of synchronous, polychronous and asynchronous models (such as CAOS or SHIM), they propose an extended intermediate representation, that is *clocked guarded actions* [6] where one can declare explicitly a set of clocks. They also show how clocked guarded actions can be used for verification by symbolic model checking (SMV) and simulation by SystemC. [7] presents an embedding of polychronous programs into synchronous ones. The embedding gives us access to the methods and tools that already exist for synchronous specifications.

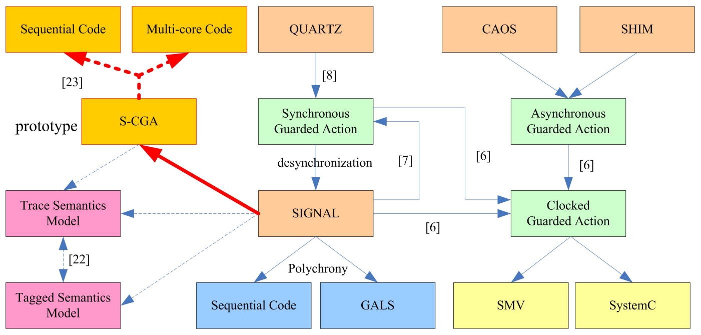

Fig. 1. A global view of the relation between our work and related work

For a safety-critical system, it is required that the compiler must be verified to ensure that the source program semantics is preserved. Our work mainly focuses on the SIGNAL language. We would like to extract a verified SIGNAL compiler from a correctness proof developed within the theorem prover Coq as it has been done in the GENEAUTO project for a part of the SIMULINK compiler. Our intermediate representation is a variant of clocked guarded actions (called S-CGA), and currently the target is multi-core code. In [23], we have already presented the compilation of sequential code and the proof of semantics preservation of the transformation from the kernel SIGNAL to S-CGA. There exist several semantics for SIGNAL, such as denotational semantics based on traces (called trace semantics) [11], denotational semantics based on tags which puts forward a partial order view of time (called tagged model semantics) [11], structural operational semantics defining inductively a set of possible transitions

[4, 11], etc. In [22], we have studied the equivalence between the trace semantics and the tagged model semantics, to assert a determined and precise semantics of the SIGNAL language. The relation between our work and related work is shown in Fig. 1.

The contribution of this paper is to propose a build method of time-predictable system on multi-core, based on synchronous-model development. At the modeling level, synchronous programming is a good choice for time-predictable system design. At the compiler level, the verified compiler from the synchronous language SIGNAL to our intermediate representation (S-CGA, a variant of guarded actions) and thus to multi-threaded code, preserves the time predictability. At the platform level, we propose a time-predictable multi-core architecture model in AADL (Architecture Analysis and Design Language) [15], and then we map the multi-threaded code to this model.

The rest of this paper is structured as follows. Section 2 presents the abstract syntax and the semantics of S-CGA. Section 3 gives the multi-threaded code generation schema from S-CGA. The time-predictable multi-core architecture model and the mapping from multi-threaded code to that model are presented in Section 4. Section 5 gives some concluding remarks.

2 S-CGA

In papers such as [6], clocked guarded actions has been defined as a common representation for synchronous (via synchronous guarded actions), polychronous and asynchronous (via asynchronous guarded actions) models. It has a multi-clocked feature. However, in contrast to the SIGNAL language, clocked guarded actions can evaluate a variable even if its clock does not hold [6] for supporting the asynchronous view. Since we focus on the polychronous view, we introduce S-CGA, which is a variant of clocked guarded actions. S-CGA constrains variable accesses as done by SIGNAL. In this section, we first present the syntax of S-CGA, and then we give the denotational semantics of S-CGA based on the trace model.

S-CGA has the same structure as clocked guarded actions, but they have *different semantics.*

Definition 1 (S-CGA). *A S-CGA system is represented by a set of guarded actions of the form $\langle \gamma \Rightarrow \mathcal{A} \rangle$ defined over a set of variables X. The Boolean condition γ is called the guard and $\mathcal{A}$ is called the action. Guarded actions can be of the following forms:*

$$\begin{array}{lll} (1) & \gamma \Rightarrow x = \tau & \text{(immediate)} \\ (2) & \gamma \Rightarrow next(x) = \tau & \text{(delayed)} \\ (3) & \gamma \Rightarrow \ assume(\sigma) & \text{(assumption)} \end{array}$$

where

- *the guard γ is a Boolean condition over the variables of X, their respective clocks (for a variable $x \in X$, we denote its clock $\hat{x}$), and their respective initial clocks (denoted $init(\hat{x})$),*

– *τ is an expression over X,*
– *σ is a Boolean expression over the variables of X and their clocks.*

An immediate assignment $x = \tau$ writes the value of τ immediately to the variable x. The form (1) implicitly imposes that if γ is defined[1] and its value is true, then x is present and τ is defined. Moreover, $init(\hat{x})$ exactly holds the first instant when x is present.

A delayed assignment $next(x) = \tau$ evaluates τ in the given instant but changes the value of the variable x at next time clock $\hat{x}$ ticks.

The form (3) defines a constraint. It determines a Boolean condition which has to hold when γ is defined and true. All the execution traces must satisfy this constraint. Otherwise, they are ignored.

Guarded actions are composed by using the parallel operator $\|$.

An S-CGA example [2] (Example 1) is shown as follows.

$true \Rightarrow assume(\widehat{y_1} = \hat{x})$	$\hat{z} \wedge z \Rightarrow s_1 = f(y_1)$
$init(\widehat{y_1}) \Rightarrow y_1 = 1$	$\widehat{s_2} \Rightarrow s_2 = s_1 + 1$
$\widehat{y_1} \Rightarrow next(y_1) = x$	$\widehat{s_1} \Rightarrow assume(\widehat{s_2})$
$true \Rightarrow assume(\widehat{y_2} = \hat{x})$	$\widehat{s_3} \Rightarrow assume(\hat{z} \wedge (not\ z))$
$init(\widehat{y_2}) \Rightarrow y_2 = 2$	$\hat{z} \wedge (not\ z) \Rightarrow s_3 = f(y_2)$
$\widehat{y_2} \Rightarrow next(y_2) = x$	$\widehat{s_4} \Rightarrow s_4 = s_3 + 2$
$true \Rightarrow assume(\hat{x} = \hat{z})$	$\widehat{s_3} \Rightarrow assume(\widehat{s_4})$
$\widehat{s_1} \Rightarrow assume(\hat{z} \wedge z)$	

Definition 2 (Trace semantics of S-CGA). *The trace semantics of a S-CGA system is defined as a set of traces, that is $[\![SCGA]\!] = \{S \mid \forall scga \in SCGA, [\![scga]\!]_S = true\}$. We have the following semantics rules,*

$$
\begin{aligned}
&(1)\quad [\![\gamma \Rightarrow x = \tau]\!]_S = \\
&\qquad \forall i \in \mathbb{N}, \widehat{[\![\gamma]\!]}_{S,i} \wedge [\![\gamma]\!]_{S,i} \\
&\qquad\quad \to (\widehat{[\![x]\!]}_{S,i} \wedge \widehat{[\![\tau]\!]}_{S,i} \wedge [\![x]\!]_{S,i} = [\![\tau]\!]_{S,i})
\end{aligned}
$$

$$
\begin{aligned}
&(2)\quad [\![\gamma \Rightarrow next(x) = \tau]\!]_S = \\
&\qquad \forall i_1 < i_2 \in \mathbb{N}, \\
&\qquad ((\forall i' \in \mathbb{N},\ i_1 < i' < i_2 \to \neg\widehat{[\![x]\!]}_{S,i'})\ \wedge \widehat{[\![\gamma]\!]}_{S,i_1} \wedge [\![\gamma]\!]_{S,i_1}) \\
&\qquad\quad \to (\widehat{[\![x]\!]}_{S,i_1} \wedge \widehat{[\![\tau]\!]}_{S,i_1} \wedge (\widehat{[\![x]\!]}_{S,i_2} \to [\![x]\!]_{S,i_2} = [\![\tau]\!]_{S,i_1}))
\end{aligned}
$$

$$
\begin{aligned}
&(3)\quad [\![\gamma \Rightarrow\ assume(\sigma)]\!]_S = \\
&\qquad \forall i \in \mathbb{N}, \widehat{[\![\gamma]\!]}_{S,i} \wedge [\![\gamma]\!]_{S,i} \to \widehat{[\![\sigma]\!]}_{S,i} \wedge [\![\sigma]\!]_{S,i}
\end{aligned}
$$

– Rule (1): when γ is present, and the value of γ is true, x and τ are both present, and the value of x is that of τ.
– Rule (2): when γ is present and the value of γ is true at instant i_1, x and τ are present at i_1, and if i_2 is the next instant where x is present, then the value of x at i_2 is that of τ at instant i_1.

[1] An expression is said to be defined if all the variables it contains are present.
[2] If two guarded actions update the same variables, the guards must be exclusive.

– Rule (3): when γ is present, and the value of γ is true, σ is present and true.

The semantics of S-CGA composition is defined as $[\![scga_1 \parallel scga_2]\!]_S = [\![scga_1]\!]_S \wedge [\![scga_2]\!]_S$.

In [23], we have already presented the translation rules from the kernel SIGNAL to S-CGA, and give the proof of the semantics preservation in Coq.

3 From S-CGA to Multi-threaded Code

The SIGNAL compilation process contains one major analysis called *clock calculus* from which *code generation* directly follows. Moreover, the clock calculus contains several steps, such as the synchronization of each process, i.e., an equation system over clocks; the resolution of the system of clock equations; the construction of a clock hierarchy on which the automatic code generation strongly relies. Our goal here is to adapt the clock calculus to S-CGA.

Based on the semantics of S-CGA, we can get the equation system over clocks. The general rules are given as follows.

S-CGA	Clock Equations
$\gamma \Rightarrow x = \tau$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{x} \wedge \hat{\tau}$
$\gamma \Rightarrow next(x) = \tau$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{x} \wedge \hat{\tau}$
$\gamma \Rightarrow assume(\sigma)$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{\sigma} \wedge \sigma$
	$init(\hat{x}) \rightarrow \hat{x}\ (\forall x \in X)$

As a first step, we just consider the *endochrony* property [3], namely we can construct a clock hierarchy based on the resolution of the system of clock equations. The clock hierarchy of Example 1 (with three clock equivalence classes C0, C1, and C2) is shown in Fig. 2. In the figure, for instance *clk_x* denotes $\hat{x}$.

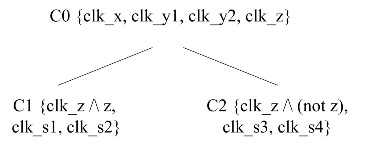
C0 {clk_x, clk_y1, clk_y2, clk_z}
C1 {clk_z /\ z, clk_s1, clk_s2}
C2 {clk_z /\ (not z), clk_s3, clk_s4}

Fig. 2. Clock hierarchy

Moreover, we construct the data-dependency graph (DDG, as shown in Fig. 3) based on the variables reading and writing.

Finally, the multi-threaded code generation is based on both the clock hierarchy and the data dependency graph. First, we map the guarded actions to threads (i.e. partitions, as shown in Fig. 3). As presented in Fig. 4, we would like to treat the partition methods generally, this means different partition methods (such as the vertical way [2] for a concurrent execution, the horizontal way [3]

[3] The weak endochrony [14] property will be considered in the future.

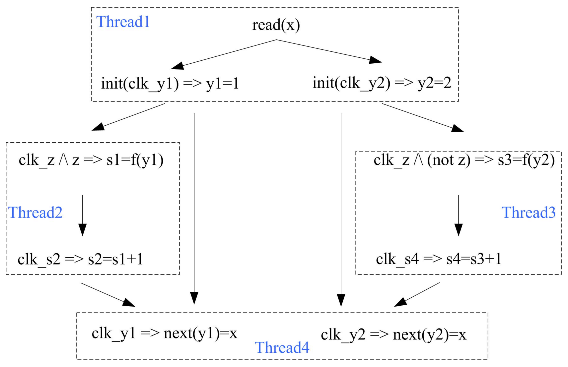

Fig. 3. Data dependency graph

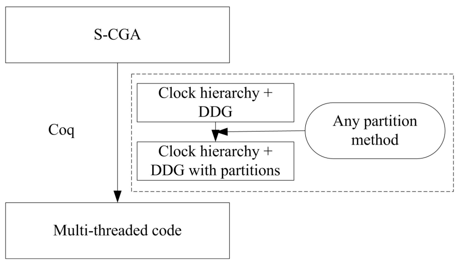

Fig. 4. The proof idea

for a pipelined execution, etc) don't affect the proof (here we don't consider performance). Second, in each thread, we organize the guarded actions based on the clock hierarchy. For example, the two guards in Thread2 belong to the same clock equivalence class, so they are merged inside the same control condition in the generated code. Third, we add wait/notify synchronization among the threads. A code fragment of Thread2 is given as follows.

```
/* Thread 2 */
void step()
{
  wait(Thread1);
  if(C1){
      $s_1 = f(y_1)$;
      $s_2 = s_1 + 1$; }
  notify(Thread4);
}
```

4 Mapping Multi-threaded Code to Multi-core

To allow for static prediction of the system timing, we need time-predictable processor architectures, thus we know all the architecture details such as the pipeline and the memory hierarchy to analyze the execution time of programs.

Furthermore, the mapping from multi-threaded code to multi-core architectures should be also static and deterministic.

4.1 A time-predictable multi-core architecture model

With the advent of multi-core architectures, interference between threads on shared resources further complicates analysis. There are some recommendations from R. Wilhelm et al. [21, 20], i.e., the better way is to reduce the time interference: (1) pipeline with static branch prediction and with in-order execution; (2) separation of caches (instruction and data caches); (3) LRU (Least Recently Used) cache replacement policy; and (4) access of main memory via a TDMA (Time Division Multiple Access) scheme. In the EC funded project T-CREST [4], M. Schoeberl et al. [18, 17] propose a new form of organization for the instruction cache, named *method cache* (MC), and split data caches (including *stack cache* (SC), *static data cache* (SDC), *constants data cache* (CDC), and *heap allocated data cache* (HC)), to increase the time predictability and to tighten the WCET. The method cache stores complete methods and cache misses occur only on method invocation and return. They split the data cache for different data areas, thus data cache analysis can be performed individually for the different areas. In our work, heap is avoided to be used because we don't use dynamic memory allocation in our multi-threaded code.

Based on these existing work, we would like to model a time-predictable multi-core architecture in AADL. AADL is an SAE (Society of Automotive Engineers) architecture description language standard for embedded real-time systems, and supports several kinds of system analysis such as schedulability analysis. Moreover, we have already worked on the semantics of different AADL subsets such as [24]. So we envision how to validate semantically the mapping from the language level to the architecture level.

Our multi-core architecture model is illustrated in Fig. 5. Inside the core, we consider static branch prediction and in-order execution in the pipeline. A simplified instruction set (*get_instruction*, *compute*, *write_data*, and *read_data*) is used. As a first step, we just consider a first level cache (i.e. without L2 and L3). Each core is associated with a method cache, a stack cache, a static data cache, and a constants data cache. However, the same principle of cache splitting can be applied to L2 and L3 caches. The extension of the timing analysis for a cache hierarchy is straight forward. Moreover, TDMA-based resource arbitration allocates statically-computed slots to the cores.

As proposed by [9], a core is associated with an AADL *processor* component and a multi-core processor with an AADL *system* component containing multiple AADL processor subcomponents, each one representing a separate core. This modeling approach provides flexibility: an AADL system can contain other components to represent cache, and shared bus, etc. For that purpose, we define specific modeling patterns with new properties (such as *Multi_Core_Properties*). A part of AADL specification is given in Fig. 6.

[4] Time-predictable Multi-Core Architecture for Embedded Systems

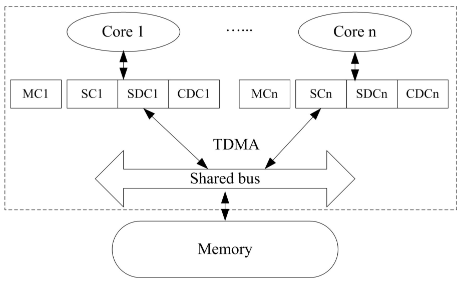

Fig. 5. A time-predictable multi-core architecture model

```
processor core
features
 MC: requires bus access cache_bus;
 SC: requires bus access cache_bus;
 SDC: requires bus access cache_bus;
 CDC: requires bus access cache_bus;
end core;

processor implementation core.impl
properties
 Multi_Core_Properties::Branch_Prediction => Static;
 Multi_Core_Properties::Execution_Order => In_Order;
end core.impl;

system multicore
features
 ExtMem: provides bus access shared_bus.impl;
end multicore;

system implementation multicore.impl
subcomponents
 Core1: processor core.impl{Multi_Core_Properties::Core_Id => 1;};
 Core2: processor core.impl{Multi_Core_Properties::Core_Id => 2;};
 Cache_MC1: memory method_cache.impl{Multi_Core_Properties::MC_Id => 1;};
 Cache_MC2: memory method_cache.impl{Multi_Core_Properties::MC_Id => 2;};
 ...
 SBus: bus shared_bus.impl;
 C2CBus1: bus cache_bus;
 C2CBus2: bus cache_bus;
connections
 bus access C2CBus1 -> Core1.MC;
 bus access C2CBus2 -> Core2.MC;
 bus access SBus -> Cache_MC1.Cache_Bus;
 bus access SBus -> Cache_MC2.Cache_Bus;
 ...
end multicore.impl;
```

Fig. 6. A part of the AADL specification of the time-predictable multi-core architecture

4.2 The mapping method

To preserve the time predictability, we consider static mapping and scheduling. Take the example shown in the last section. It generates a configuration file (such as *num_of_threads=4*) in multi-threaded code generation. Moreover, we have a manual configuration file for the time-predictable multi-core architecture model, for example *num_of_cores=4*. Thus, we can generate a static mapping and scheduling, for instance:

- Thread1 $\mapsto$ Core1, Thread2 $\mapsto$ Core2, Thread3 $\mapsto$ Core3, and Thread4 $\mapsto$ Core4.
- Thread1: notify(Thread2), notify(Thread3);
  Thread2: wait(Thread1), notify(Thread4);
  Thread3: wait(Thread1), notify(Thread4);
  Thread4: wait(Thread2), wait(Thread3).

Based on the simplified instruction set (considered in the architecture model), the multi-core code can be generated. Thanks to the mechanizations such as method cache, split data caches, TDMA and static scheduling, the execution time of the multi-core code can be bounded.

5 Conclusion and Future Work

With the widespread advent of multi-core processors in safety-critical embedded systems or cyber-physical systems (CPS), it further aggravates the complexity of timing analysis. This paper proposes a build method of time-predictable system on multi-core, based on synchronous-model development. Our method integrates time predictability across several design layers, i.e., synchronous programming, verified compiler, and time-predictable multi-core architecture model. Interaction among cores might also arm software isolation layers, such as the one defined in ARINC653. Thanks to the existing work such as [9] and [13] on AADL modeling on multi-core architectures and their association with ARINC653, we would like to associate our work with partitioned architectures in the future.

Acknowledgments: This work was supported by the RTRA STAE Foundation in France (http://www.fondation-stae.net/).

References

1. P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, and W. Yi. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, 13(4):82:1–82:37, Mar. 2014.
2. D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. DATE '10, pages 949–952, 2010.
3. D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Generating software pipelines for OpenMP. In M. Dietrich, editor, *MBMV*, pages 11–20. Fraunhofer Verlag, 2010.
4. A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
5. F. Boussinot and R. de Simone. The esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

6. J. Brandt, M. Gemünde, K. Schneider, S. K. Shukla, and J.-P. Talpin. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. *Design Automation for Embedded Systems*, pages 1–35, 2012.
7. J. Brandt, M. Gemunde, K. Schneider, S. K. Shukla, and J.-P. Talpin. Embedding polychrony into synchrony. *IEEE Trans. Software Eng.*, 39(7):917–929, 2013.
8. J. Brandt and K. Schneider. Separate translation of synchronous programs to guarded actions. *Internal Report 382/11, Department of Computer Science, University of Kaiserslautern*, 2011.
9. J. Delange and P. Feiler. Design and analysis of multi-core architecture for cyber-physical systems. In *5th Embedded Real Time Software and Systems*, ERTS'14, February 2014.
10. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
11. A. Gamatié. *Designing embedded systems with the SIGNAL programming language.* Springer, 2010.
12. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
13. J. Hugues. AADLib, a library of reusable AADL models. In *SAE Aerotech 2013 Congress & Exhibition (Montreal, Canada)*, September 2013.
14. D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006.
15. SAE. AS5506A: Architecture Analysis and Design Language (AADL) Version 2.0. 2009.
16. K. Schneider. The synchronous programming language quartz. *Internal report, Department of Computer Science, University of Kaiserslautern, Germany*, 2010.
17. M. Schoeberl. A time predictable instruction cache for a Java processor. In R. Meersman, Z. Tari, and A. Corsaro, editors, *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, volume 3292 of *Lecture Notes in Computer Science*, pages 371–382. Springer, 2004.
18. M. Schoeberl, B. Huber, and W. Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 49(1):1–28, 2013.
19. J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S. Shukla. Constructive polychronous systems. In *Logical Foundations of Computer Science*, volume 7734 of *Lecture Notes in Computer Science*, pages 335–349. Springer, 2013.
20. L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Syst.*, 28(2-3):157–177, Nov. 2004.
21. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem: Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.
22. Z. Yang, J.-P. Bodeveix, and M. Filali. A comparative study of two formal semantics of the SIGNAL language. *Frontiers of Computer Science*, 7(5):673–693, 2013.
23. Z. Yang, J.-P. Bodeveix, M. Filali, H. Kai, and D. Ma. A verified transformation: From polychronous programs to a variant of clocked guarded actions. In *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, SCOPES '14, pages 128–137. ACM, 2014.
24. Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, and J.-P. Talpin. From AADL to timed abstract state machines: A verified model transformation. *Journal of Systems and Software*, 93:42–68, 2014.