Correct-by-construction code generation from hybrid automata specification

Davide Bresolin, Luigi Di Guglielmo, Luca Geretti, and Tiziano Villa
Dipartimento di Informatica – Università di Verona, Italy
{davide.bresolin, luigi.diguglielmo, luca.geretti, tiziano.villa}@univr.it

***Abstract*—In the last years hybrid automata have been applied in the design and verification of embedded systems. Once a hybrid model of the system has been proved to be correct with respect to the desired properties, it would be valuable to extract a *correct-by-construction* HW/SW implementation of it. This work discusses a methodology and a corresponding tool chain that allow to extract a HW/SW implementation of a controller modeled by a subclass of timed automata, named *elastic controllers*, operating in an environment represented by a hybrid automaton. The required tools have been either developed from scratch or extended from the current state-of-the-art in order to support an automated flow from hybrid automata specifications to correct-by-construction discrete implementations described in the SystemC language.**

***Index Terms*—hybrid automata; code generation; SystemC; verification; implementability; embedded systems;**

I. Introduction

When designing embedded systems, often the need arises to model systems having a mixed discrete and continuous behaviour that cannot be characterized faithfully using either a discrete or continuous model only. An example is an automotive powertrain system, where a four-stroke engine is modelled by a switching continuous system and is controlled by a digital controller. Such systems consist of a discrete control part that operates in a continuous environment and are named cyberphysical or hybrid systems because of their mixed nature. Hybrid automata are a powerful formalism for the design and verification of such embedded systems [1], since they allow to describe, in the earlier design phase, the functional and quantitative temporal aspects of the continuous and discrete components, and can be composed together generating a model of the whole system that can be formally verified. However, while the problems of modeling and verifying networks of hybrid systems are extensively studied in the literature, much less work has been done in developing techniques to automate the subsequent phases of the design flow. Once a hybrid model of the system has been proved to be correct with respect to the desired properties, it would be valuable to extract a correct by construction HW/SW implementation of it. This is called a refinement phase: given a high-level description of the system, refine it into another description such that all the "important" properties of the original one are preserved.

Even though many tools support automatic code generation from the hybrid model (for example, Matlab Simulink [2], Ptolemy [3]), the emphasis has been on performance-related optimizations, and many issues relevant to correctness are not satisfactorily addressed. First, the precise relationship between the model and the generated code is rarely specified or formalized. Second, the continuous blocks are either ignored, or discretized before code generation [4]. Finally, code generation typically means generation of tasks, and does not incorporate scheduling. Consequently, the correspondence between the model and the code is broken, and formal verification results established for the model are not meaningful for the code.

This work proposes a complete framework for refinement from the hybrid down to the discrete domain of embedded systems, originally specified by a network of hybrid automata modeling the embedded system and its continuous environment. By a correct-by-construction procedure (relying on the theory of *Almost-ASAP* semantics [5]) we generate code in the SystemC language and compute performance bounds to be satisfied by any conservative concrete hardware implementation. This refinement methodology provides the first complete platform for the embedded design community to experiment with automatic code generation of a system specified with hybrid automata. This enabling technology realized with open source code is a first step to address the open problems both in theory (how to extend the class of hybrid controllers that can be synthesized automatically?), and in practice (what is the most "efficient" transformation from the hybrid to the discrete domain to favour scalability?).

The paper is organized as follows. Section II summarizes the related works. Section III describes the proposed methodology for code generation from the specification of hybrid automata. Section IV describes two case studies on which the methodology has been applied. Finally, Section V is related to concluding remarks.

II. Related Works

Only few works in literature focus on the correct-by-construction extraction of an implementation starting from high level hybrid models [6].

In [5] an alternative semantics for timed automata is proposed to extract from a timed model C of a digital component a correct by construction discrete implementation $C_{\mathcal{D}}$ in a systematic way. The new semantics, named Almost-ASAP, takes into account the digital and imprecise aspects of the hardware in which the hybrid automaton $\mathcal{C}$ is being executed, i.e. it relaxes the instantaneousness and the perfect precision typical of the hybrid models, allowing the authors to identify

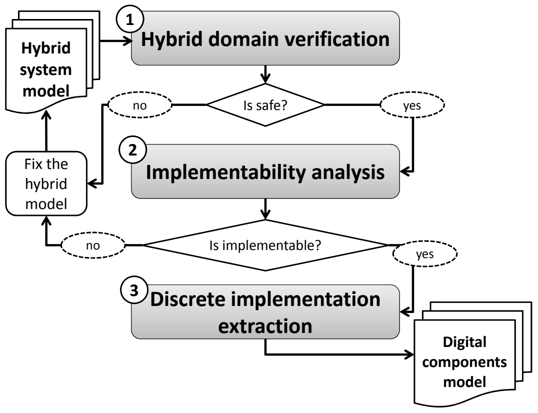

Fig. 1. Methodology overview.

a relaxed hybrid model $C_{\mathcal{R}}$ that - once formally verified - guarantees the existence of a discrete implementation. Such an implementation consists of a program that, executed into any platform with a certain worst-case-execution-time and with a sufficiently precise clock, refines correctly the original model. Unfortunately, this methodology uses an ad hoc language to model the hybrid system and, moreover, the generated code can be executed only on a toy OS, i.e., BrickOS [7].

Notice that modifying the semantics may not be the only way to enforce the implementability. Indeed, in [8] the authors ask the question whether similar results can be obtained without introducing a new semantics, but acting on modeling instead, thanks to the introduction of new assumptions on the program type or execution platform by means of changes, in a modular way, on the corresponding models. The authors propose an implementation methodology for timed automata which allows to transform a timed automaton into a program and to check whether the execution of this program on a given platform satisfies a desired property. Unlike the work in [5], an open problem of this approach is how to guarantee that, when a platform P is replaced by a "better" platform P', a program proved correct for P is also correct for P'; the authors reported examples where this does not hold for a reasonable assumption of a "better" platform, namely, when P and P' are identical, but P' provides a periodic digital clock running twice as fast as the one of P; the reason is that a program using the faster clock has a higher "sampling rate" and thus may generate more behaviors than a program using the slower clock, so this situation may result in a violation of properties.

III. Formal Verification and Refinement Methodology

The developed methodology for extracting a correct-by-construction discrete HW/SW implementation, starting from a hybrid model of the system (Figure 1), is divided into three phases:

1) *Formal verification of the hybrid system*. This phase consists of formally verifying the correctness of the hybrid model M of the system, composed of a digital controller $\mathcal{C}$ and a continuous environment $\mathcal{E}$, against the set $\mathcal{P}$ of properties that the system should respect, i.e., $\mathcal{C}$ is able to safely handle $\mathcal{E}$ according to the set $\mathcal{P}$ of properties.
2) *Implementability analysis*. This phase consists of determining if a discrete implementation $\mathcal{C}_{\mathcal{D}}$ of the digital controller can preserve the functional and temporal behaviors of the original hybrid model $\mathcal{C}$, i.e., $\mathcal{C}_{\mathcal{D}}$ is able to safely handle $\mathcal{E}$ according to the set $\mathcal{P}$ of properties.
3) *Extraction of the discrete implementation*. This phase consists of refining the hybrid model of the digital controller $\mathcal{C}$ by extracting a discrete implementation $\mathcal{C}_{\mathcal{D}}$ which captures the functional and temporal aspects of its original hybrid model.

The novel aspects of the proposed methodology are described thoroughly in the following sections.

A. Implementability Analysis

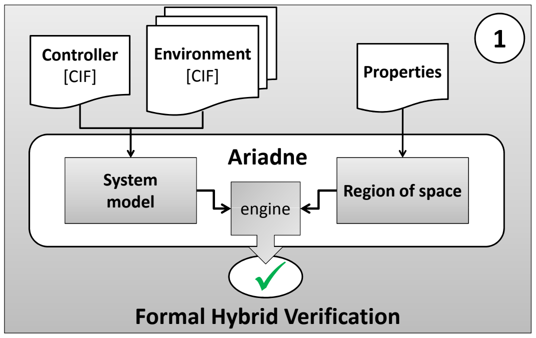

Fig. 2. Formal Hybrid Verification flow.

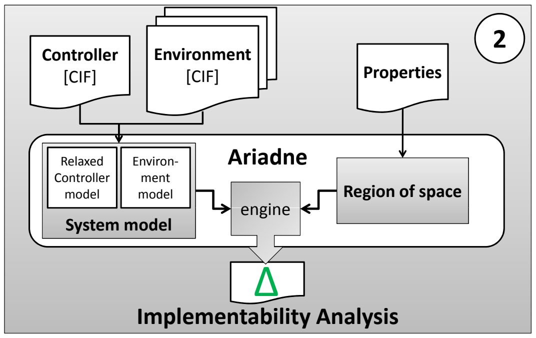

Fig. 3. Implementability Analysis flow.

Once the hybrid model M has been proved to be correct by formal hybrid verification (Figure 2), i.e., the controller $\mathcal{C}$ safely handles the environment $\mathcal{E}$ under the assumption of infinite variables precision and zero-delays typical of the hybrid domain, it is possible to proceed with the analysis of the system implementability (Figure 3). This analysis faces the problem of the semantic gap between the hybrid models and the discrete implementations. In particular:

- *infinite precision*. Variables of hybrid automata take their values from a dense set (i.e. $\mathbb{R}$), whereas variables in the hardware domain are always discrete and thus have limited precision;

- *instantaneousness*. Communication between the interacting components of the hybrid model is instantaneous, whereas communication in real implementations always introduces delays and it is mandatory to make sure that the control strategy remains correct when the latter happen.

For these reasons, a model that has been proved correct in the traditional semantics may not be implementable (at all), or it may not be possible to turn it automatically into an implementation that is correct by construction [9].

As a consequence, an implementability analysis must take into account the digital and imprecise aspects of the hardware on which the discrete model of the digital component is being executed. In particular, it concerns:

- *the relaxation of the variable precision*: continuous variables can be modeled only with a finite precision and consequently they are rounded according to the HW platform characteristics;
- *the relaxation of the instantaneousness of reaction to timeouts and events*: any reaction to a timeout and an incoming or outgoing event introduces delays that depend on the HW platform characteristics.

By considering these relaxations, it is possible to determine if the hybrid model of a digital component can be refined into a discrete implementation. This *relaxed model* is characterized by a parameter δ such that:

- *δ relaxes the continuous variable precision*. The guards of the transitions that involve the evaluation of the continuous variable are parameterized by δ, simulating variable rounding;
- *δ relaxes the reactions to timeouts*. Any transition that can be taken by the automaton becomes urgent after a small delay modeled by the parameter δ;
- *δ relaxes the reactions to events*. A distinction is made between the occurrence of an event in the sender (occurrence) and the acknowledgement of the event by the receiver (perception). The time difference between the occurrence and the perception of the event is bounded by δ.

By considering such modifications to the original hybrid model of a digital component, which are consistent with the Almost-ASAP semantics [5], it is possible to determine how much the model behavior can be relaxed while preserving system correctness. Notice that the relaxed model exhibits a superset of the original behavior, the latter corresponding to a relaxed value $\Delta = 0$ for the parameter δ. Consequently, in order for the relaxed model to be implementable at all, a necessary condition is that there exists a value $\Delta > 0$ for which the required properties are satisfied (Figure 1).

Currently, the proposed approach is valid on a subclass of timed automata, called *elastic controllers*, featuring the following restrictions:

1) Only urgent transitions are allowed;
2) The guards must be closed expressions;
3) Communication with the environment is allowed only through events.
4) Continuous variables are restricted to clocks (i.e. continuous variables measuring the elapsing of time).

It must be remarked that the restrictions above are perfectly reasonable from the controller implementation viewpoint and do not represent a major limitation in terms of applicability of the method.

To perform the refinement analysis, we automatically extract from $\mathcal{M}$ a new model $\mathcal{M}'$ that, when proved to be correct against the hybrid system specifications, guarantees the existence of a discrete implementation that refines the original hybrid model. The model $\mathcal{M}'$ is given by the composition of the continuous environment $\mathcal{E}$ and the relaxed controller $\mathcal{C}_{\mathcal{R}}$ that is obtained from $\mathcal{C}$ by applying the modifications summarized above. After its generation, $\mathcal{M}'$ is verified against the set $\mathcal{P}$ of original properties. The verification results provide the maximum value Δ for δ that returns a relaxed safe system. Given Δ, the expression $\Delta > 4\Delta_P + 3\Delta_L$ proved in [5] relates it to the actual constraints of the discrete implementation such as the clock period (Δ_P) and communication latency (Δ_L). It holds that, if Δ represents a strictly positive value for δ, then a discrete implementation $\mathcal{C}_{\mathcal{D}}$ such Δ_P and Δ_L satisfy the above expression is able to control the continuous environment $\mathcal{E}$ respecting the original properties, and, moreover, it can be refined automatically from $\mathcal{C}$ as described in the following section.

B. *Extracting the Discrete Implementation*

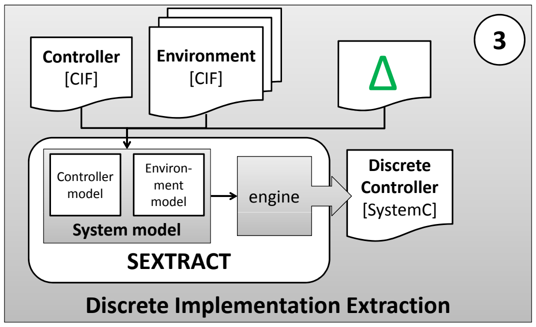

Fig. 4. Discrete Implementation Extraction flow.

Finally, once the relaxed model turns out to be implementable, it is possible to proceed with the discrete implementation extraction phase (Figure 4) that consists of extracting behaviors from the hybrid model in such a way that the code implementing the digital components (e.g., the controller $\mathcal{C}$) is correct by construction.

In order to allow the correct acknowledgement of input events and the emission of output events without missing the synchronization with the continuous components, the generated discrete implementation is composed of several threads:

- a *main thread* with period Δ_P that implements the functional aspects of the original hybrid model, in which the

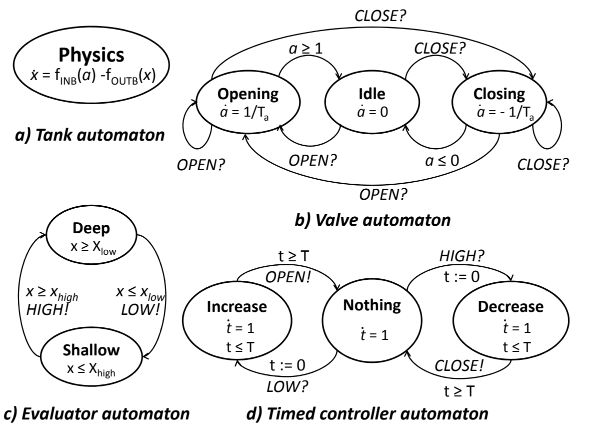

Fig. 5. Hybrid model of the watertank system.

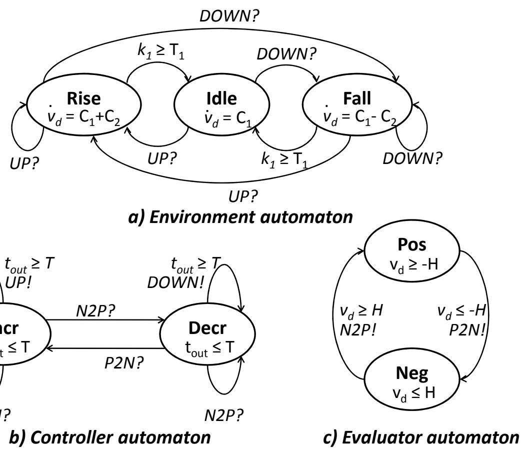

Fig. 6. Hybrid model of the Power Supply Selector system.

guards of the transitions related to timeouts are annotated with the temporal constraint Δ so that no active transition is missed due to clocks rounding;

- many *input handler threads* (one for each input event) aiming (in a Δ_L time units window) to catch and dispatch to the main thread the input events coming from the environment, in such a way that even asynchronous inputs can be correctly detected by the periodic main thread.

The discrete model obtained in this way is described by means of SystemC [10]. SystemC is the de facto ESL (Electronic System Level) language based on C++. Thus, unlike plain sequential C code, its adoption allows to handle real-time HW/SW components at different abstraction levels (TLM and RTL) guaranteeing that the generated description can be used as a reference model for the discrete refinement phase. Indeed, the SystemC simulation and verification environment gives the possibility to exploit current existing assertion-based verification methodologies to validate the discrete implementation at each step of the subsequent discrete refinement process.

IV. Case Study

The methodology described in this work has been validated on two hybrid systems: the watertank system [11], and the Power Supply Selector (PSS) system that comes from an industrial example [12].

The watertank system is depicted in Figure 5, where four different automata are shown: a tank, a valve, an evaluator and a timed controller. Briefly, the system is centered on a water tank, which is characterized by an uncontrolled outbound water flow, while the inbound water flow is controlled by the aperture of a valve. The controller acts on the aperture of the valve a in order to keep the water level x in a safe interval $x_{min} < x < x_{max}$.

The hybrid model depicted in Figure 6 represents the Power Supply Selector (PSS) included in the MAGALI platform [12]. The basic behavior of the PSS is to control the supply voltage V_c of a generic unit of the platform. More precisely, due to DVFS (Dynamic Voltage and Frequency Scaling) operations, the supply voltage can switch dynamically between two values, *High* and *Low*. During such transitions, the supply voltage V_c (which supplies the considerable load given by the equivalent resistance of the core circuit) must follow a linearly rising/-falling reference voltage V_r as closely as possible. Essentially, a controller provides periodic UP or $DOWN$ events that ultimately make the supply voltage rise or fall by a fixed step of voltage: by ensuring that the controller issues events at a properly high frequency, the core voltage can follow the reference voltage, guaranteeing a bounded voltage difference V_d between V_r and V_c.

A. Formal Safety Verification

The first step of the methodology consists of verifying the two systems against the properties they should satisfy. To check the watertank system we used the following simple property $\varphi = always(x < x_{max} \ \& \ x > x_{max})$ that requires that the water level x is always kept between the safe bounds $x_{max} = 8.25$ and $x_{min} = 5.25$ (while $x_{high} = 8.0$ and $x_{low} = 5.5$). To check the PSS system, instead, the property chosen was $\psi = always(V_d >= -L \ \& \ V_d <= L)$, meaning that the voltage difference $V_d = V_r - V_c$ must maintain an absolute value not greater than $L = 0.1$. Thanks to the adoption of the Compositional Interchange Format (CIF) [13] for modeling the two hybrid systems, it has been possible to interface to already existing verification tools for hybrid models such as Ariadne [14]. Ariadne can read a CIF description of the system in which each component is modeled as a separate automaton composed with the others by using parallel composition. This notably simplifies the system design due to the fact that the composed system is automatically generated from the single automata by the internal engine of the tool. Ariadne can check safety reachability properties that are internally converted into a region of space: the system is safe as long as its reached region lies inside the safe region. Due to the conservative rounding of values, the reached region is provided as an over-approximation of the actual region; it must be noticed that the quality of the approximation depends on the evolution and discretisation parameters chosen. In particular, while low quality can prevent verification in some cases, high quality necessarily requires a longer verification time. For this reason, Ariadne implements an iterative mechanism for verification: it starts using a low-quality approximation and refines it until an answer to the verification problem is obtained. If a positive answer is ultimately found, then the system is safe in the ideal case of infinite clock precision and zero-delays.

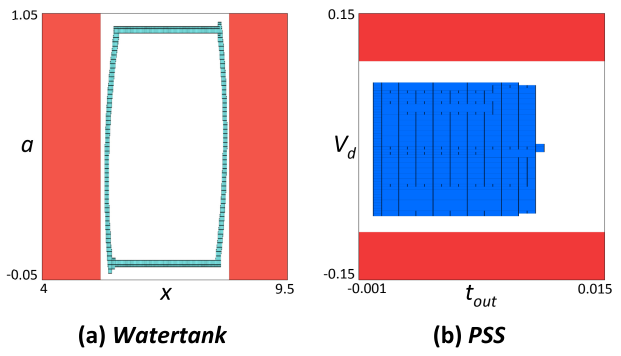

Fig. 7. (a) Watertank: projection on the x-a plane of the reachable region (in the center) and of the unsafe region (on left and right), where x is the water level and a is the valve aperture. (b) PSS: projection on the t_{out}-V_d plane of the reachable region (in the center) and of the unsafe region (on top and bottom), where t_{out} is the controller clock and V_d is the voltage difference.

Figure 7 shows the results of the reachability analysis performed by Ariadne, respectively for (a) the watertank and (b) the PSS systems. Notice that an overapproximation of the systems evolution is shown after discretization as a set of boxes in the center of the figures, while the unsafe regions are the ones represented at the left and right extremes in (a) and at the top and bottom extremes in (b). Since the (overapproximated) reachable regions do not intersect the unsafe regions, the systems are safe.

B. Implementability Analysis

Once the hybrid models were proved to be correct, we performed the subsequent implementability analysis phase. Notice that we extended the internal engine of Ariadne to realize automatically the transformations that determine the implementability of a hybrid model. Thus, after loading the hybrid models into Ariadne, the tool generated the corresponding relaxed models and provided the maximum value Δ that sets bounds on the relaxed safe systems. For the watertank system, the tool returned an upper bound for Δ equal to *0.289898872375*, for which the relation $\Delta > 4\Delta_P + 3\Delta_L$ must hold. For example, if our platform is characterized by a communication latency upper bounded by *1 ms* (i.e. $\Delta_L \leq 1/1000$ sec.), the clock frequency of the controller must be greater or equal than *14 Hz* (i.e. $\Delta_P \leq 1/14$ sec.). A choice of Δ_P and Δ_L for which the relation above does not hold would possibly cause either a failure in capturing input events, or a failure in providing output events with sufficient promptness. Which of these two cases is ultimately responsible for an undesired behavior of the environment actually depends on the temporal evolution of the system. For the PSS system, the tool retrieved an upper bound for Δ equal to *0.00112487792969*. Again, if for example the communication delay is upper bounded by *60* μs, then the controller must guarantee a clock frequency not slower than *10 Mhz* (i.e., a clock precision of at least *10* μs).

C. Discrete Implementation Extraction

Once the controllers embedded into the two systems are verified as being implementable, by using Sextract it is possible to extract their discrete implementations. We developed Sextract from scratch in such a way that it reads the system implementation (i.e. the CIF model of the controller $\mathcal{C}$ and the environment $\mathcal{E}$) and the value Δ synthesized during the implementability analysis phase, and extracts the functional behaviors of the hybrid model of $\mathcal{C}$ annotated with the corresponding temporal constraint Δ. Such an implementation provides a refinement of the functionalities of the timed controller that allows the correct acknowledgement of input events and the emission of output events, coming from and destined to the environment respectively.

The structure of the implementations (Listing 1 and 2) is characterized by (i) a main thread that models the controller behavior (e.g. *WTcontroller::automaton()* and *PSScontroller::automaton()*) and (ii) support threads that handle the incoming events (e.g. *WTcontroller::check_HIGH()* and *PSScontroller::check_N2P()*) notifying them to the related main thread. The execution of such threads is completely managed by the SystemC simulation kernel [10].

About the controller, it is worth noting how its model is refined: (i) the guards of the transitions related to timeouts are annotated with the temporal constraint Δ to identify correctly the transitions active in presence of discrete clocks (e.g. Listing 1 line 21), (ii) support functions are used to handle clocks values (i.e. *get/set* functions for handling the rounding) and (iii) incoming events are detected by reading the value of the variable set by the related event-handler thread (e.g. *check_HIGH()* and *check_LOW()*).

These implementations described by means of SystemC can be used as they are or can be wrapped into SystemC TLM components. Thus, it is possible to use one of the already existing methodologies for going on with the discrete refinement phase. Moreover, the adoption of SystemC also allows to exploit assertion-based verification approaches described in literature to check the correctness of the subsequent refined implementations.

```
1  #include "../inc/controller.h"
2  const double controller::T = 0.1;
3  const double controller::delta = 0.289898872375;
4  void WTcontroller::automaton() {
5    local_modes automaton_mode = nothing;
6    while (true) {
7      wait();
8      if (automaton_mode == nothing) {
9        if (LOW_pending) {
10         LOW_read.write(!LOW_read.read());
11         set_clock_value(t, 0);
12         automaton_mode = increase;
13       } else if (HIGH_pending) {
14         HIGH_read.write(!HIGH_read.read());
15         set_clock_value(t, 0);
16         automaton_mode = decrease;
17       }
```

```
    } else if (automaton_mode == increase) {
      tcp(is_le(get_clock_value(t), 0.1));
      if (is_ge(get_clock_value(t), 0.1 - delta)) {
        OPEN.write(!OPEN.read());
        automaton_mode = nothing;
      }
    } else if (automaton_mode == decrease) {
      tcp(is_le(get_clock_value(t), 0.1));
      if (is_ge(get_clock_value(t), 0.1 - delta)) {
        CLOSE.write(!CLOSE.read());
        automaton_mode = nothing;
      }
    }
  }
}
void WTcontroller::check_HIGH() {
  if (HIGH.event()) {
    HIGH_pending = true;
  } else if (HIGH_read.event()) {
    HIGH_pending = false;
  }
}
void WTcontroller::check_LOW() {
  if (LOW.event()) {
    LOW_pending = true;
  } else if (LOW_read.event()) {
    LOW_pending = false;
  }
}
```

Listing 1. The discrete model of the Controller of the Watertank system.

```
#include "../inc/Controller.h"
const double Controller::T = 0.01;
const double Controller::delta = 0.00112487792969;
void PSScontroller::automaton() {
  local_modes automaton_mode = Incr;
  while (true) {
    wait();
    if (automaton_mode == Incr) {
      tcp(is_le(get_clock_value(t), 0.01));
      if (is_ge(get_clock_value(t), 0.01 - delta)) {
        UP.write(!UP.read());
        set_clock_value(t, 0);
        automaton_mode = Incr;
      } else if (P2N_pending) {
        P2N_read.write(!P2N_read.read());
        automaton_mode = Incr;
      } else if (N2P_pending) {
        N2P_read.write(!N2P_read.read());
        automaton_mode = Decr;
      }
    } else if (automaton_mode == Decr) {
      tcp(is_le(get_clock_value(t), 0.01));
      if (is_ge(get_clock_value(t), 0.01 - delta)) {
        DOWN.write(!DOWN.read());
        set_clock_value(t, 0);
        automaton_mode = Decr;
      } else if (N2P_pending) {
        N2P_read.write(!N2P_read.read());
        automaton_mode = Decr;
      } else if (P2N_pending) {
        P2N_read.write(!P2N_read.read());
        automaton_mode = Incr;
      }
    }
  }
}
void PSScontroller::check_N2P() {
  if (N2P__event()) {
    N2P_pending = true;
  } else if (N2P_read__event()) {
    N2P_pending = false;
  }
}
void PSScontroller::check_P2N() {
  if (P2N__event()) {
    P2N_pending = true;
  } else if (P2N_read__event()) {
    P2N_pending = false;
  }
}
```

Listing 2. The discrete model of the Controller of the PSS system.

V. Concluding Remarks

The development of techniques for the extraction of correct-by-construction HW/SW implementations of hybrid systems is a new and valuable research area. This work proposes a complete design flow for the extraction of functional aspects and temporal constraints from hybrid systems in order to obtain a discrete implementation. To support the methodology, we built a tool chain which includes Ariadne, a hybrid automata verifier extended in order to support the discretization semantics, and Sextract, a new tool able to extract the necessary constraints from the system and ultimately generate its discrete implementation code. To the best of our knowledge, this is the first example of a complete automatic flow that goes from a hybrid model down to a SystemC implementation of it.

Acknowledgments

This research was partly supported by the EU projects FP7-IST-1-217069 COCONUT and FP7-ICT-223844 CON4COORD.

References

[1] T. Henzinger, "The Theory of Hybrid Automata," in *IEEE Symposium on Logic in Computer Science (LICS)*, 1996, pp. 278 – 292.

[2] The MathWorks, Inc., "Simulink 7.6," http://www.mathworks.com, 2010.

[3] Center for Hybrid and Embedded Software Systems (CHESS), University of California at Berkeley, "Ptolemy II," http://ptolemy.berkeley.edu/ptolemyII/.

[4] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "TIMES - A Tool for Modelling and Implementation of Embedded Systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2002, pp. 460–464.

[5] M. Wulf, L. Doyen, and J. Raskin, "Almost ASAP Semantics: from Timed Models to Timed Implementations," *Formal Aspects of Computing*, vol. 17, no. 3, pp. 319 – 341, 2005.

[6] R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky, "Generating Embedded Software from Hierarchical Hybrid Models," in *ACM SIGPLAN Conference*, vol. 38-7, 2003, pp. 171–182.

[7] M. Noga, "BrickOS," http://brickos.sourceforge.net.

[8] K. Altisen and S. Tripakis, "Implementation of Timed Automata: an Issue of Semantics or Modeling?" in *Formal Modeling and Analysis of Timed Systems (FORMATS)*, 2005, pp. 273–288.

[9] F. Cassez, T. Henzinger, and J. F. Raskin, "A Comparison of Control Problems for Timed and Hybrid Systems," in *Hybrid Systems: Computation and Control (HSCC)*, 2002, pp. 134–148.

[10] Open SystemC Initiative, "SystemC," 1999, http://www.systemc.org.

[11] L. Benvenuti, A. Ferrari, E. Mazzi, and A. Vincentelli, "Contract-based Design for Computation and Verification of a Closed-loop Hybrid System," *Hybrid Systems: Computation and Control*, pp. 58–71, 2008.

[12] E. Beigné, F. Clermidy, S. Miermont, P. Vivet, and G. MINATEC, "Dynamic Voltage and Frequency Scaling Architecture for Units Integration within a GALS NoC," in *ACM/IEEE International Symposium on Networks-on-Chip (NoCS)*, 2008, pp. 129–138.

[13] C. Sonntag, R. Schiffelers, D. van Beek, J. Rooda, and S. Engell, "Modeling and Simulation using the Compositional Interchange Format for Hybrid Systems," in *International Conference on Mathematical Modelling (MATHMOD)*, 2009, pp. 640–650.

[14] A. Balluchi, A. Casagrande, P. Collins, A. Ferrari, T. Villa, and A. Sangiovanni-Vincentelli, "Ariadne: a Framework for Reachability Analysis of Hybrid Automata," in *International Symposium on Mathematical Theory of Networks and Systems (MTNS)*, 2006.