ezRealtime: A Domain-Specific Modeling Tool for Embedded Hard Real-Time Software Synthesis

Fabiano Cruz, Raimundo Barreto, Lucas Cordeiro
Departamento de Ciência da Computação
Universidade Federal do Amazonas
{fcruz,rbarreto,lcc}@dcc.ufam.edu.br

Paulo Maciel
Centro de Informática (CIn)
Universidade Federal de Pernambuco
prmm@cin.ufpe.br

Abstract

*In this paper, we introduce the **ezRealtime** project, which relies on the Time Petri Net (TPN) formalism and defines a Domain-Specific Modeling (DSM) tool to provide an easy-to-use environment for specifying Embedded Hard Real-Time (EHRT) systems and for synthesizing timely and predictable scheduled C code. Therefore, this paper presents a generative programming method in order to boost code quality and improve substantially developer productivity by making use of automated software synthesis. The ezRealtime tool reads and automatically translates the system's specification to a time Petri net model through composition of building blocks with the purpose of providing a complete model of all tasks in the system. Hence, this model is used to find a feasible schedule by applying a depth-first search algorithm. Finally, the scheduled code is generated by traversing the feasible schedule, and replacing transition's instances by the respective code segments. We also present the application of the proposed method in an expressive case study.*

1 Introduction

This work considers Embedded Hard Real-Time (EHRT) software development. Regarding real-time systems, the correct behavior depends not only on the integrity of the results, but also the time in which such results are produced.

In this paper, we adopted Time Petri net formal model for modeling the system and finding a feasible schedule in order to prove that it exists. However, for the effective use of formalisms, an important issue to be considered is the availability of an abstraction layer, through which developers can model their application without necessarily knowing that there is an underlying formal semantics .

Domain-Specific Modeling (DSM) environments are intended to automate the creation of program parts that are costly to build from scratch. It is a graphical representation of a Domain-Specific Language (DSL) that is targeted to a particular matter, rather than a general purpose language that can be used to develop all kinds of programs. Therefore, we created the ezRealtime[1] tool, which provides a DSM Language (DSML) based on a time Petri net formalism and a code generator engine with the purpose of automating several parts of the development of EHRT softwares. Therefore, the proposed work aims at developing an open source DSM environment to provide not just a friendly GUI from where all system's functionalities can be specified, but also a generative programming approach to boost code quality and improve developer productivity with automated software synthesis.

A DSML is one of the most suitable approach to deal with today's software complexity with high abstraction levels. There are some DSL building frameworks used to speed up the development process, such as GME[7], Microsoft DSL Tools[5], and Eclipse Modeling Project Platform[2].

2 Related Works

We have identified other projects that also consider this important subject.

The TOPCASED project[12] relies on the Eclipse Modeling Project Platform, and the metamodeling principle is the core of this project. A new DSL is proposed, namely SimplePDL, which is an experimental language for specifying processes. It introduces a temporal extension of OCL, TOCL, based on process states and formalized using a LTL (Linear Temporal Logic). Furthermore, Petri nets are also used to model checking purposes.

Sztipanovits and Karsai[11] discuss challenges and opportunities of generative programming (developing programs that synthesize other programs) for embedded software development. It explains the the principles of MIC

[1] http://pnmp.sourceforge.net/ezrealtime/
[2] EMP, http://www.eclipse.org/modeling/

978-3-9810801-3-1/DATE08 © 2008 EDAA

(Model-Integrated Computing), which places models as center piece for the integrated software development.

These works are very close to what we achieved in this work. Indeed, ezRealtime also combines a operational semantics based on timed Petri nets with DSL Engineering. In addition, ezRealtime provides an easy-to-use model-based software development environment for modeling, checking properties, generating schedule, and synthesizing code.

3 Modeling

3.1 Computational Model

Computational model syntax is given by a time Petri net [9], and its semantics by a timed labeled transition system (TLTS) which uses a time discrete model. A time Petri net (TPN) is a bipartite directed graph represented by a tuple $\mathcal{P}= (P,T,F,W,m_0,I)$. P (places) and T (transitions) are non-empty disjoint sets of nodes. The edges are represented by $F \subseteq (P\times T)\cup(T\times P)$. $W : F \to \mathbb{N}$ represents the weight of the edges. A TPN marking m_i is a vector $m_i \in \mathbb{N}^{|P|}$, and m_0 is the initial marking. $I : T \to \mathbb{N}\times\mathbb{N}$ represents the timing constraints, where $I(t) = (EFT(t), LFT(t))\ \forall t \in T$, $EFT(t) \leq LFT(t)$, $EFT(t)$ is the Earliest Firing Time, and $LFT(t)$ is the Latest Firing Time. An extended time Petri net with code and priorities is represented by $\mathcal{P}_a = (\mathcal{P}, \mathcal{CS}, \pi)$. $\mathcal{P}$ is the time Petri net, $\mathcal{CS}$:$T \nrightarrow \mathcal{ST}$ is a partial function that assigns transitions to behavioral source code, where $\mathcal{ST}$ is a set of source tasks codes, and $\pi : T \to \mathbb{N}$ is a priority function.

The set of states S of $\mathcal{P}$ is given by $S \subseteq (M \times C)$, where each state is defined by a marking, and its clock vector.

$FT(s)$ is the set of fireable transitions at state s defined by: $FT_P(s) = \{t_i \in ET(m) \mid \pi(t_i) = \min(\pi(t_k)) \wedge DLB(t_i) \leq \min(DUB(t_k)),\ \forall t_k \in ET(m)\}$. The *firing domain* for t at state s, is defined by the interval: $FD_s(t) = [DLB(t), \min(DUB(t_k))]$.

The semantics of a TPN $\mathcal{P}$ is defined by associating a TLTS $\mathcal{L}_{\mathcal{P}}= (S,\Sigma,\to,s_0)$: (i) S is the set of states of $\mathcal{P}$; (ii) $\Sigma \subseteq (T\times\mathbb{N})$ is a set of actions labeled with (t,θ) corresponding to the firing of transition (t) at time (θ) in the firing interval $FD_s(t)$, $\forall s \in S$; (iii) $\to \subseteq S\times\Sigma\times S$ is the transition relation; (iv) s_0 is the initial state of $\mathcal{P}$.

Definition 3.1 (Reachable States) *Let $\mathcal{L}_{\mathcal{P}}$ be a TLTS derived from a TPN $\mathcal{P}$, and $s_i = (m_i, c_i)$ a reachable state. s_{i+1} =fire$(s_i,(t,\theta))$ denotes that firing a transition t at time θ from the state s_i, a new state $s_{i+1} = (m_{i+1}, c_{i+1})$ is reached, such that: (1) $\forall p \in P$, $m_{i+1}(p) = m_i(p) - W(p,t) + W(p,t)$; (2) $\forall t_k \in ET(m_{i+1})$: (i) $C_{i+1}(t_k) = 0$ (if $(t_k = t) \vee (t_k \in ET(m_{i+1}) - ET(m_i))$), or (ii) $C_{i+1}(t_k) = C_i(t_k) + \theta$, otherwise.*

Definition 3.2 (Feasible Firing Schedule) *Let $\mathcal{L}_{\mathcal{P}}$ be a TLTS derived from a TPN $\mathcal{P}$, s_0 its initial state, $s_n = (m_n, c_n)$ a final state, and $m_n = M^F$ is the desired final marking. $s_0 \xrightarrow{(t_1,\theta_1)} s_1 \xrightarrow{(t_2,\theta_2)} s_2 - - \to s_{n-1} \xrightarrow{(t_n,\theta_n)} s_n$ is defined as a feasible firing schedule, where s_i = fire$(s_{i-1}, (t_i,\theta_i))$, $i > 0$, if $t_i \in FT(s_{i-1})$, and $\theta_i \in FD_{s_{i-1}}(t_i)$.*

The modeling methodology guarantees that the final marking M^F is well-known since it is explicitly modeled.

3.2 Specification Model

The proposed specification model is composed by: (i) a set of tasks with timing constraints; (ii) intertask relations; (c) the schedule method for each task (preemptive or non-preemptive), and the behavioral specification.

Let $\mathcal{T}$ be the set of tasks in a system. The proposed approach considers only periodic tasks, where the definition of timing constraints is as follows. Let $\tau_i \in \mathcal{T}$ be a periodic task. The constraints of τ_i is defined by $(ph_i, r_i, c_i, d_i, p_i)$, where ph_i is the phase offset time; r_i is the release time; c_i is the worst-case execution time (WCET); d_i is the deadline; and p_i is the period.

The phase (ph_i) is the delay associated to the first time request of task τ_i after the system starting. The periodicity in which τ_i is requested is denoted by the period p_i. Release time r_i, WCET c_i, and deadline d_i, are time instants considering the beginning of the period as the start point. Thus, r_i is the earliest time where the task τ_i may start execution, c_i is the WCET required for executing task τ_i; and d_i is the time at which task τ_i must be completed. This work considers that $c_i \leq d_i \leq p_i$.

The considered inter-tasks relations are precedence and exclusion relations. A task τ_i PRECEDES task τ_j, if τ_j can only start executing after τ_i has finished. A task τ_i EXCLUDES task τ_j, if no execution of τ_j can start while task τ_i is executing, i.e., task τ_i could not be preempted by task τ_j. Exclusion relations may prevent simultaneous access to shared resources. We consider symmetrical exclusion relation, that is, if A EXCLUDES B then B EXCLUDES A.

The behavioral specification consists of the source code for each task. This code is programmed using the C programming language, and it must be in accordance with the respective compiler for the target processor.

3.3 Modeling the Specification

This section details how to model the specification using time Petri net formal model through composition of building blocks. It is worth observing that such blocks are specific for the pre-runtime scheduling policy.

The proposed modeling method is conducted by building block compositions. This work adopts several operators for

building block compositions. Details about such operators is beyond the scope of this paper. The interested reader is referred to [2].

Pre-runtime scheduling considers the entire set of periodic tasks occurring within a time period that is equal to the least common multiple (LCM) among periods of the given set of tasks. The LCM is also called schedule period (PS) or hyper-period. Therefore, there are several tasks instances of the same task within the schedule period.

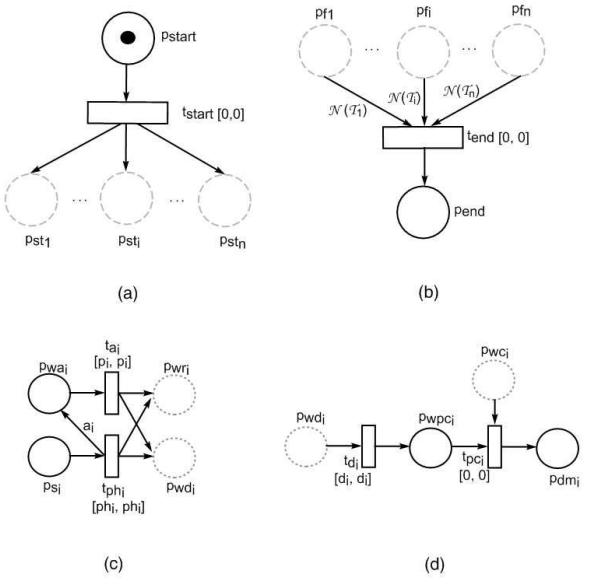

Figure 1. Proposed Blocks - Part 1

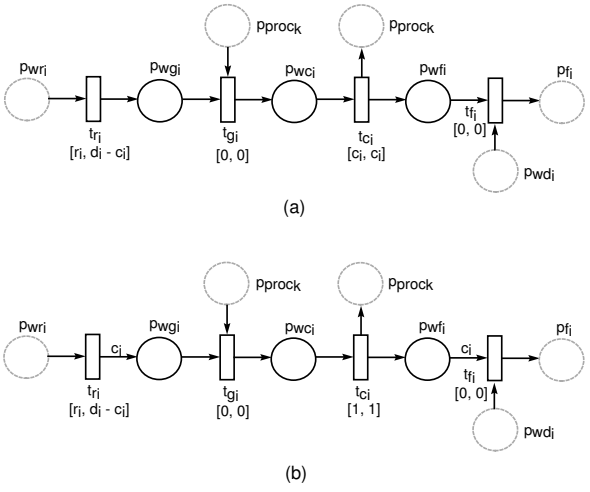

Figure 2. Proposed Blocks - Part 2

3.3.1 Building Blocks

Tasks are modeled by composition of building blocks depicted in Figures 1 and 2, and summarized below: a) **Fork Block**. The fork block (Fig. 1(a)) models the starting of n concurrent tasks. The timing interval of transition t_{start} is always equal to [0, 0]; b) **Join Block**. The join block (Fig. 1(b)) models the fact that all n tasks have concluded their execution in the schedule period. In the proposed join block, $m_i(p_{end}) = 1$ indicates that a feasible firing schedule (Def. 3.2) was found; c) **Periodic Task Arrival Block**. This block (Fig. 1(c)) models the periodic invocation of all instances of all tasks in the schedule period (PS). It is worth noting the weight $(\alpha_i = \mathcal{N}(\tau_\rangle) - \infty)$ of the arc (t_{ph_i}, p_{wa_i}), where this weight models the invocation of all remaining instances after the first task instance. The timing intervals of transitions t_{a_i} and t_{ph_i} are fulfilled by ph_i (phase) and p_i (period) of task τ_i; **d) Deadline Checking Block**. Some works (e.g. [1]) extended the Petri net model for dealing with deadline checking. The proposed modeling method uses elementary net structures to capture deadline missing. Obviously, Deadline missing (Fig. 1(d)) is an undesirable situation when considering hard real-time systems. The timing interval for transition t_{pc_i} is constant, and for transition t_{d_i} is fulfilled by the deadline d_i of task τ_i. e) **Non-preemptive Task Structure Block**. Considering a non-preemptive scheduling method, the processor is just released after the entire computation has been finished. Figure 2(a) shows that time interval of computation transition has bounds equal to the task computation time (i.e., $[c_i, c_i]$). The timing interval for transition t_{g_i} is constant, and for transitions t_{r_i} and t_{c_i} are fulfilled by release r_i, and execution time c_i of task τ_i. f) **Preemptive Task Structure Block**. This scheduling method (Fig. 2(b)) implies that a task is implicitly split into subtasks, where the computation time of each subtask is exactly equal to one time unit. The timing of transition t_{r_i} is fulfilled by r_i (release) of task τ_i. All remaining timing intervals are constants; **g) Processor Block**. This work is constrained to mono processor architecture. Hence, as the processor is considered as a resource, the processor block consists of a single place p_{proc} with one marking. This modeling is important since the processor is used in a mutually exclusive way.

3.3.2 Inter-tasks Relations Modeling

Inter-tasks relations are modeled as follows:

a) Modeling Precedence Relations. Precedence relations are defined between pairs of tasks. Let us suppose that τ_i `PRECEDES` τ_j is specified. After modeling the two tasks (τ_i and τ_j), represented by nets N_i and N_j, respectively, some actions are performed in order to model such precedence relation. Figure 3 shows a TPN model representing a precedence relation. It worth observing that task T_2 can

only proceed after task T_1 has finished its execution.

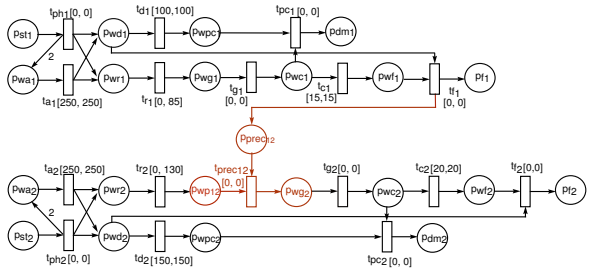

Figure 3. Precedence Relation Model

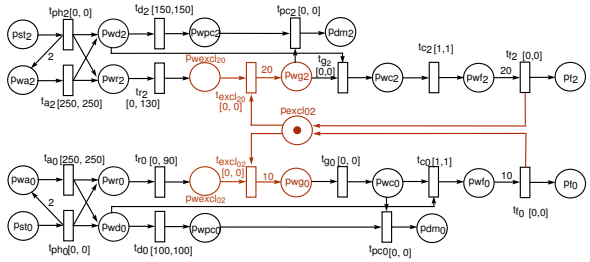

Figure 4. Exclusion Relation Model

b) Modeling Exclusion Relations. Exclusion relations are also defined between pairs of tasks. Let us suppose that τ_i EXCLUDES τ_j is specified. The modeling method adds a single place shared by the two tasks. This place has one marking and it is pre-condition for the execution of the two tasks. Therefore, just one of both tasks is executing simultaneously. After modeling the two tasks (τ_i and τ_j), represented by nets N_i and N_j, respectively, some actions are performed to model the exclusion relation: Fig. 4 shows a TPN model representing an exclusion relation.

4 ezRealtime: The EHRT Modeling Tool

4.1 Project Overview

The ezRealtime is an open source project which relies on the time Petri net formalism and defines a Domain Specific Modeling Language (DSML) to provide an easy-to-use environment for specifying embedded hard real-time systems and for synthesizing timely and predictable scheduled C code. It uses the International Standard ISO/IEC 15909-2[6] which defines a universal XML-based transfer syntax for Petri nets, namely Petri Net Markup Language (PNML)[13].

ezRealtime is distributed under the Apache License version 2.0 and it has been developed using Eclipse Modeling Project Platform, in particular the Eclipse Modeling Framework (EMF) [3]. EMF plays an important role in this work and it has proved to be a mature tool that offers a straightforward approach to develop DSLs. EMF is a Java framework and code generation facility for building tools and other applications based on a metamodel. Once the metamodel for a particular domain is specified, the EMF can generate a set of Java code, including Eclipse plug-ins, and graphical/customizable editors. EMF metamodels can be defined as an UML class diagram, Annotated Java interfaces with some model properties, XML Schema Definitions (XSDs), or directly in a XMI document.

4.2 Metamodeling

Modeling describes the concepts of a domain with the concepts provided by a modeling language. Metamodeling explores the use of modeling languages. Thus, it allows the definition of tailored or DSM languages. ezRealtime uses the EMF to transform the proposed specification metamodel, represented as a UML class diagram, into Ecore (see Figure 5). For lack of space, the entire metamodel is not shown in this paper.

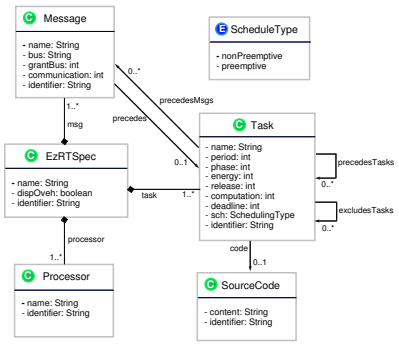

Figure 5. Specification Metamodel

4.3 The tool architecture of ezRealtime

In order to develop the **ezRealtime**, we created a DSML based on a time Petri net formalism and the EMF. *Codegen* code generator facility translates this model into a tree view graphical editor, where end-users define a set of tasks and their inter-relations, which is in turn transformed into a human and machine readable PNML (a XML-based document markup standard) description of the application. This PNML file is built in compliance with the proposed Building Blocks, Operators, and Net Compositions approaches. Furthermore, it serves as basis for the pre-runtime **ezRealtime** scheduler engine that is used to find a feasible schedule, and then C code is automatically synthesized.

In the proposed tool, end-users do not need to know that there is an underlying formal semantics that provide the basis for the automation of software synthesis. Therefore, it ensures that system's properties are satisfied and the system is properly validated according to the specification. The tool architecture of **ezRealtime** is illustrated in Figure 6.

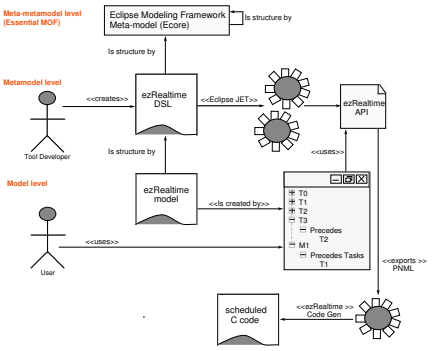

Figure 6. tool architecture of ezRealtime

In order to generate a model from the specification, the following steps should be taken into account: i) generate a model for arrival, deadline, and task structure blocks for each task; ii) generate each precedence and exclusion relations; iii) generate each inter-tasks communication; iv) generate the fork block; and v) generate the join block.

The ezRealtime uses its transformation engine (a domain-specific component library) with a third-party API called PNML Framework[3] for mapping from ezRealtime DSL (see Figure 7) into timed Petri nets through the PNML (ezRealtime2PNML).

```
<?xml version="1.0" encoding="UTF-8"?>
<rt:ez-spec xmlns:rt="http://pnmp.sf.net/EZRealtime">
  <Task precedesTasks="#ez1151891690363" identifier="ez1151891690
    <processor>p124365</processor>
    <name>T1</name>
    <period>9</period>
    <power>10</power>
    <schedulingMode>NP</schedulingMode>
    <computing>1</computing>
    <deadline>9</deadline>
  </Task>
...
</rt:ez-spec>
```

Figure 7. ezRealtime DSL

4.4 Code Generator Engine

A ezRealtime CodeGen library was developed in order to automate the code generation process. Such engine uses the Ruby[4] programming language to generate code for microcontrollers. The next subsections are concerned with describing the scheduler synthesis and the scheduled code generator.

4.4.1 Pre-Runtime Schedule Synthesis

Scheduling is very important in embedded real-time systems. The proposed scheduler synthesis algorithm is a depth-first search method on a finite timed labeled transition system derived from a TPN model. The algorithm may experience the state explosion problem when searching for a feasible schedule. In order to keep the state space growth under control, the proposed method adopts a partial-order minimization technique [8] in order to prune the state space. The proposed algorithm is a depth-first search method on a generated timed labeled transition system (TLTS). The *stop criterion* is obtained whenever the desirable final marking M^F is reached. The algorithm is explained in our previous works presented in [2].

4.4.2 Scheduled Code Generation

The proposed method for code generation includes not only tasks' code, but also a timer interrupt handler, and a small dispatcher. Such dispatcher automates several control mechanisms required during the execution of tasks. Timer programming, context saving, context restoring, and tasks' calling are examples of such additional controls. An array of registers (`struct ScheduleItem`) is created to store the schedule table. Each input represents the *execution part* of a task instance. In case of preemption, a task instance may have more than one *execution part*. The register `struct ScheduleItem` contains the following information: (i) start time; (ii) flag, indicating if the task was preempted before; (iii) task id; and (iv) a pointer to a function (task code). Figure 8 depicts the schedule table for a preemptive application. It includes two instances of `TaskA`, two instances of `TaskB`, two instances of `TaskC`, and one instance of `TaskD`.

```
struct ScheduleItem scheduleTable [SCHEDULE_SIZE] =
{{ 1, false, 1, (int *)TaskA}, /* A1 starts */
 { 4, false, 2, (int *)TaskB}, /* B1 preempts A1 */
 { 6, false, 3, (int *)TaskC}, /* C1 preempts B1 */
 { 8, true,  2, (int *)TaskB}, /* B1 resumes */
 {10, false, 4, (int *)TaskD}, /* D1 preempts B1 */
 {11, true,  2, (int *)TaskB}, /* B1 resumes */
 {13, true,  1, (int *)TaskA}, /* A1 resumes */
 {18, false, 1, (int *)TaskA}, /* A2 starts */
 {20, false, 3, (int *)TaskC}, /* C2 preempts A2 */
 {22, false, 2, (int *)TaskB}, /* B2 starts */
 {28, true,  1, (int *)TaskA}  /* A2 resumes */
};
```

Figure 8. Example of a Schedule Table

5 Case Study: Mine System

This case study is a real-world application, where detailed specification for this example can be found in [4].

[3]PNML Framework, http://www.lip6.fr/pnml

[4]http://www.ruby-lang.org

This system is a simplified pump control system for a mining environment. The system is used to pump mine-water, collected in a sump at the bottom of the shelf to the surface. When the water reaches a given high-level the pump is turned on and the sump is drained until the water reaches the low-level. At this point, the pump is turned off. The pump should only be allowed to operate if the methane level (CH_4) in the mine is below a critical level. The monitoring also measures the level of carbon monoxide (CO) in the mine and detects whether there is as adequate flow of air.

Table 1. Specification for Mine Pump

task	Computation	Deadline	Period
PMC	10	20	80
WFC	15	500	500
RLWH	1	1000	1000
CH4H	25	500	500
CH4S	5	100	500
COH	15	100	2500
AFH	15	200	6000
WFH	15	300	500
PDL	15	500	500
SDL	10	500	500

Table 1 presents the system specification. This problem has 10 tasks, implying 782 tasks' instances and, at the beginning, all 10 tasks arrive at the same time. Our solution searched 3268 states (where minimum number of states is 3130) in 330 ms. The platform was an AMD Athlon 1800 MHz processor, with 768 MB RAM, adopting Linux operating system with GCC 4.0.2 compiler.

6 Conclusion and Future Work

ezRealtime has been designed to provide developers with an easy-to-use interface for specifying Embedded Hard Real-Time systems and for synthesizing timely and predictable scheduled C code, which can be leveraged in the applications. Such software uses transformation component library for mapping the proposed DSL into the rigorous semantics of time Petri nets.

The **ezRealtime** tool *per se* is a contribution. The more specific contributions to the DSL Engineering and EHRT domains are: *(i)* propose a formalized software modeling process using time Petri nets, *(ii)* describe a DSML that supports developers to specify EHRT systems, and also generate C code that make system deployment easier, and *(iii)* provide a tool based methodology for development of predictable scheduled code for EHRT systems.

For further steps, we are evolving the **ezRealtime** project, both to improve its *CodGen* component and tranformation rules. Our aim is to apply the proposed methodology in the development of the EHRT software for several kinds of microcontrollers and processors (e.g., ARM9, 8051, M68K, x86) in a generative way. Moreover, we also aim to optimize the generated code to specific platforms.

Acknowledgments

The authors would like to thank the support received from the Nokia Institute of Technology (INdT) and the Brazilian Agency CNPq process number: 553164/2005-8.

References

[1] K. Altisen, G. Göbler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. *IEEE Real-Time System Symposium*, pages 154–163, December 1999.

[2] R. Barreto. *A Time Petri Net-Based Methodology for Embedded Hard Real-Time Software Synthesis*. PhD Thesis, Centro de Informática - UFPE, April 2005.

[3] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[4] A. Burns and A. Wellings. HRT-HOOD: A structured design method for hard real-time systems. *Real-Time Systems Journal*, 6(1):73–114, 1994.

[5] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[6] E. Kindler. Software and systems engineering - high-level petri nets. part2: Transfert format, 2005.

[7] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing, Hungary*, volume 17, May 2001.

[8] J. Lilius. Efficient state space search for time petri nets. In *Electronic Notes in Theoretical Computer Science*, volume 18. Elsevier Science, 1998.

[9] P. Merlin and D. J. Faber. Recoverability of communication protocols: Implicatons of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043, Sept. 1976.

[10] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD Thesis, MIT, May 1983.

[11] J. Sztipanovits and G. Karsai. Generative programming for embedded systems. In *PPDP '02: Proc 4th ACM SIGPLAN conf on Principles and practice of declarative programming*, 2002.

[12] F. Vernadat, C. Percebois, P. Farail, R. Vingerhoeds, A. Rossignol, J.-P. Talpin, and D. Chemouil. The TOPCASED Project - A Toolkit in OPen-source for Critical Applications and SystEm Development. In *Data Systems In Aerospace (DASIA), Berlin, Germany, 22/05/2006-25/05/2006*. European Space Agency (ESA Publications), mai 2006.

[13] M. Weber and E. Kindler. The Petri Net Markup Language. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication Based Systems*, LNCS 2472. Springer-Verlag, 2003.