Towards a Model-Driven Engineering Approach for Developing Embedded Hard Real-Time Software

Fabiano Cruz
Nokia Institute of Technology (INdT) and Universidade Federal do Amazonas
fabianoc@acm.org

Raimundo Barreto
Departamento de Ciência da Computação
Universidade Federal do Amazonas
rbarreto@dcc.ufam.edu.br

Lucas Cordeiro
Departamento de Ciência da Computação
Universidade Federal do Amazonas
lcc@dcc.ufam.edu.br

ABSTRACT

Model-Driven Engineering (MDE) has been advocated as an effective way to deal with today's software complexity. MDE can be seen as an integrative approach combining existing techniques such as Domain-Specific Modeling Languages (DSML) and Transformation Engines. This paper presents the **ezRealtime**, an MDE-based tool that relies on the Time Petri Net (TPN) formalism and defines a DSML to provide an easy-to-use environment for specifying Embedded Hard Real-Time (EHRT) systems and for synthesizing timely and predictable scheduled C code. The ezRealtime adopts the universal XML-based transfer syntax for Petri nets, named as PNML. The main idea of this work is to propose a generative programming method and tool to boost code quality and improve developer productivity with automated software synthesis. The ezRealtime tool reads and automatically translates the 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. Therefore, 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 a case study.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Petri nets; D.3.2 [**Language Classifications**]: Specialized Application Languages—*domain specific languages, modeling languages, model transformation languages*

Keywords

Embedded Hard Real-Time Systems, Model-Driven Engineering, Petri Nets, Software Synthesis, DSL Engineering, Tool-based approaches

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.

The adoption of formal methods is an alternative to deal with several kinds of constraints that need to be satisfied, such as timing, size, weight, energy consumption, and reliability. Formal methods are important mechanisms for the analysis and verification of properties, as well as, facilitate system validation. However, for the effective use of formalisms, the availability of an abstraction layer through which developers can model their application in graphic design area, without necessarily knowing that there is an underlying formal semantics, is an important issue and needs to be considered.

Model-Driven Engineering (MDE) tools impose domain-specific constraints and perform model checking that can detect and prevent many errors early in the life cycle [13]. MDE can be seen as an integrative approach combining existing techniques such as Domain-Specific Modeling Languages (DSML) and Transformation Engines.

DSML 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. There are some DSL building frameworks used to speed up the development process, such as GME[9], Microsoft DSL Tools[5], and Eclipse Modeling Project Platform[1].

In this context, we developed the ezRealtime[2], a MDE-based tool which provides a DSML backed by a time Petri net formalism and a code generator engine with the purpose of automating several parts of the development of EHRT software. It aims at developing an open source MDE-based environment to provide not just a friendly GUI from where all system's functionalities can be specified, but a generative programming approach to boost code quality and improve developer productivity with automated software synthesis.

The rest of this work is organized as follows: Section 2 describes related works. Section 3 describes the method for modeling embedded hard real-time systems. A deep dive on project ezRealtime is the subject of Section 4. Section 5

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

showcases an experiment. Finally, Section 6 concludes this paper and depicts future works.

2. RELATED WORKS

MDE is gaining momentum in the development of complex software systems. We have identified other projects that also consider this important subject.

The TOPCASED project[15] is an open source CASE environment for critical applications and systems development. It relies on the Eclipse Modeling Project Platform and the metamodelling principles forms the core of this project. A new DSL is proposed and used as case study in this work, namely SimplePDL (Simple Process Description Language), 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). In additional to that, Petri nets are also used for model checking purposes.

Sztipanovits and Karsai[14] discusses challenges and opportunities of generative programming (developing programs that synthesize other programs) for embedded software development. It explains the principles of MIC (Model-Integrated Computing), a project from the ISIS laboratory of the Vanderbilt University, which places models as center piece for the integrated software development.

Another important work is the Times tool which is intend to support the embedded system designer for modelling, schedulability analysis, synthesis of schedules and executable code[3]. One of the main characteristics of Times tool is that it is suitable for systems that can be decomposed by a set of preemptive or non-preemptive tasks which are triggered periodically or sporadically by time or external events. Moreover, this tool provides means to interactively model and simulate the dynamic behavior of the system as well as verify the schedulability analysis. Nevertheless, this tool only allows the user to automatically generate the C-code for the brickOS platform.

These works are very close to that achieved in this work. Indeed, ezRealtime also combines a operational semantics based on timed Petri nets with DSL Engineering. In additional, it provides an easy-to-use model-based software development environment for modeling EHRT systems, schedule software synthesis and model validation engine.

3. MODELING

3.1 Computational Model

Computational model syntax is given by a time Petri net [11], and its semantics by a timed labeled transition system. 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 $F \subseteq (P \times T) \cup (T \times P)$. $W : F \rightarrow \mathbb{N}$ 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. $I : T \rightarrow \mathbb{N} \times \mathbb{N}$ 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 $\pi : T \rightarrow \mathbb{N}$ is a priority function.

A set of enabled transitions is denoted by: $ET(m_i) = \{t \in T \mid m_i(p_j) \geq W(p_j, t)\}$, $\forall p_j \in P$. The time elapsed, since the respective transition enabling, is denoted by a clock vector $c_i \in \mathbb{N}^{|ET(m_i)|}$. The dynamic firing interval $(I_D(t))$ is dynamically modified whenever the respective clock variable $c(t)$ is incremented, and t does not fire. $I_D(t)$ is computed as follows: $I_D(t) = (DLB(t), DUB(t))$, where $DLB(t) = max(0, EFT(t) - c(t))$, $DUB(t) = LFT(t) - c(t)$, $DLB(t)$ is the Dynamic Lower Bound, and $DLB(t)$ is the Dynamic Upper Bound.

DEFINITION 3.1 (STATES). *Let $\mathcal{P}$ be a time Petri net, M be the set of reachable markings of $\mathcal{P}$, and C be the set of clock vectors. The set of states S of $\mathcal{P}$ is given by $S \subseteq (M \times C)$, that is a state is defined by a marking, and the respective 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 $\mathcal{L}_{\mathcal{P}}= (S, \Sigma, \rightarrow, s_0)$: (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 a firable transition (t) at time (θ) in the firing interval $FD_s(t)$, $\forall s \in S$; (iii) $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation; (iv) s_0 is the initial state of $\mathcal{P}$.

DEFINITION 3.2 (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} $=\mathtt{fire}(s_i, (t, \theta))$ 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) $\forall p \in P$, $m_{i+1}(p) = m_i(p) - W(p, t) + W(t, p)$; (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 states that the firing of a transition t_i, at time value θ_i, in state (s_i), defines the next state (s_{i+1}).

DEFINITION 3.3 (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.*

$$s_0 \overset{(t_1,\theta_1)}{\longrightarrow} s_1 \overset{(t_2,\theta_2)}{\longrightarrow} s_2 - - \rightarrow s_{n-1} \overset{(t_n,\theta_n)}{\longrightarrow} s_n$$

is defined as a feasible firing schedule, where s_i $= \mathtt{fire}(s_{i-1}, (t_i, \theta_i))$, $i > 0$, 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 (preemptive or nonpreemptive) for each task, 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 $(ph_i, r_i, c_i, d_i, p_i)$,

[3]The Times Tool, http://www.timestool.com

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.

In real applications, there are some cases where the arrival of tasks is not periodic. These tasks are generally called aperiodic tasks, since they arrive randomly. A particular class of aperiodic tasks is called sporadic tasks. The minimum period between two activations of a sporadic task is known. However, pre-runtime approaches may only schedule periodic tasks. In order to schedule sporadic tasks, each one should be translated into an equivalent periodic task. In this paper such translation is based on Mok's work [12].

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.

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.

3.3.1 *Building Blocks*

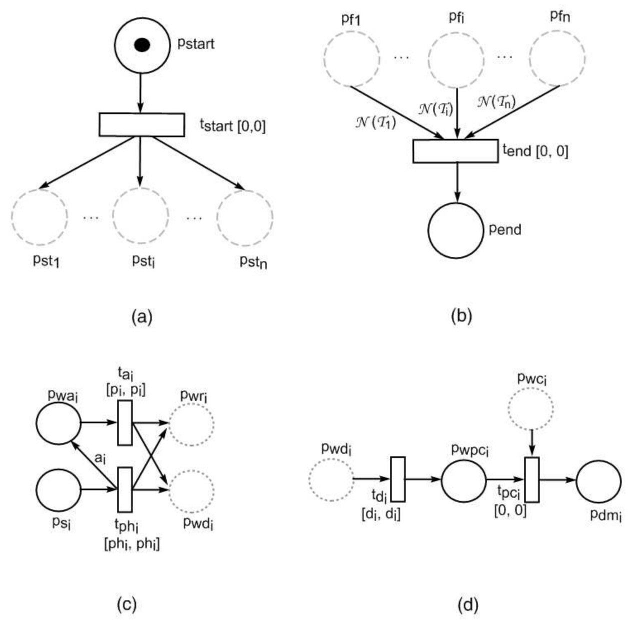

Figure 1: Proposed Building Blocks - Part I

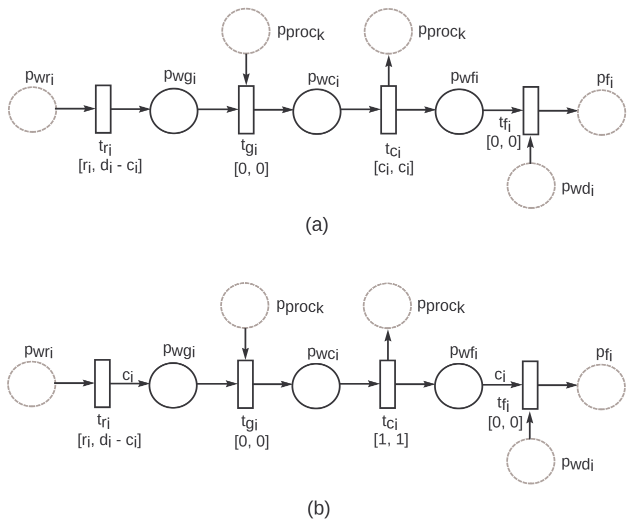

Figure 2: Proposed Building Blocks - Part II

Tasks are modeled by composition of building blocks depicted in Figures 1 and 2, and summarized below: a) **Fork Block**. The fork block models the starting of n concurrent tasks. Figure 1(a) shows the fork block. The timing interval of transition t_{start} is always equal to [0, 0]; b) **Join Block**. The join block models the fact that all n tasks have concluded their execution in the schedule period. Figure 1(b) presents the join block. It is worth noting that a marking in place p_{end} represents the desirable final marking (or M^F). In this case, $m_i(p_{end}) = 1$ indicates that a feasible firing schedule (Def. 3.3) was found; c) **Periodic Task Arrival Block**. This block models the periodic invocation of all instances of all tasks in the schedule period (PS). Figure 1(c) illustrates the periodic task arrival block. 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 (Figure 2(b)) implies that a task is implicitly split into subtasks, where the computation time of each subtask is exactly equal to one TTU. 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 Operators and Net Compositions

The proposed modeling method is conducted by building block compositions. This work proposes several operators depicted as follows: (a) *Place merging* operator is adopted to make a new net by merging two set of places of two blocks to be composed; (b) *Serial place refinement* can be seen as a replacement of a single place by a sequence of one place, one transition, and another place; (c) *Arc addition* is an operator that adds an arc from a place to a transition or from a transition to a place; (d) *Arc removing* is an operator that removes a specific arc; (e) *Place addition* is an operator that adds a single place to a net; and (f) *Net union* is an operator that joins two nets producing another net. More details about compositions and operators are beyond the scope of this paper. The interested reader is referred to [2].

3.3.3 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.

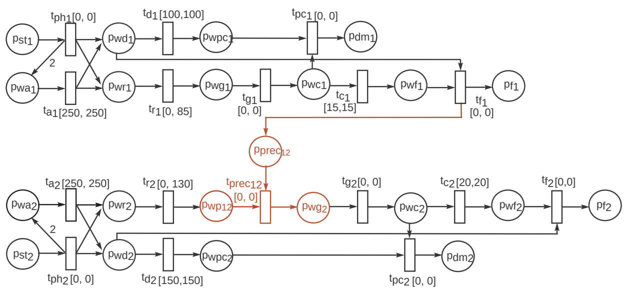

Figure 3: Precedence Relation Model

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 proposed 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: Figure 4 shows a TPN model representing a exclusion relation.

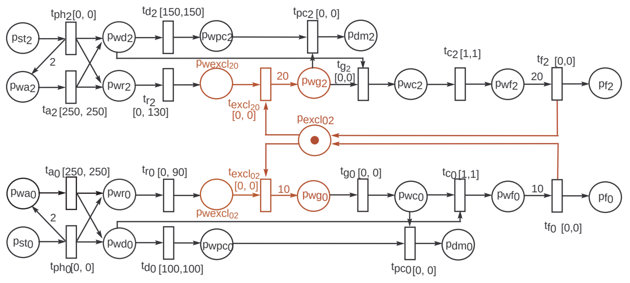

Figure 4: Exclusion Relation Model

4. DEEP DIVE ON PROJECT EZREALTIME

4.1 Project Internals

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 EHRT systems and for synthesizing timely and predictable scheduled C code. It uses the International Standard ISO/IEC 15909-2[8] which defines a universal XML-based transfer syntax for Petri nets, namely Petri Net Markup Language (PNML).

ezRealtime is distributed under the Apache License version 2.0 and it has been developed using Eclipse Modeling Project Platform, in special the Eclipse Modeling Framework (EMF) [3] that plays an important role in this work and has proved to be a mature tool that offers a simple and elegant 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 (Rational Rose or UML2), Annotate Java interfaces with some model properties, XML Schema Definitions (XSDs), or directly in a XMI document. Additionally, EMF can also be thought as a Java implementation of a core subset of the OMG standard MOF (Meta-Object Facility) [6] API. MOF is a common language across metamodels, and in its current proposal (MOF 2.0), a similar subset of the MOF model, which is called Essential MOF (EMOF), is separated out. To avoid any confusion, the MOF-like core metamodel in EMF is called Ecore. There are small differences between Ecore and EMOF; however, EMF can transparently read and write serializations of EMOF.

4.2 Metamodelling

Modeling describes the concepts of a domain with the concepts provided by a modeling language. Metamodelling allows for the modelling of modelling languages. It thus allows the definition of tailored, or DSM languages. ezRealtime uses the EMF to transform the proposed specification (Section 3.2) metamodel (represented as a UML class diagram) representation (see Figure 5) into Ecore.

EMF also allows you to go back and forth from the model to the generated code, iteratively refining the specification.

For lack of space in Figure 5, the entire metamodel is not shown.

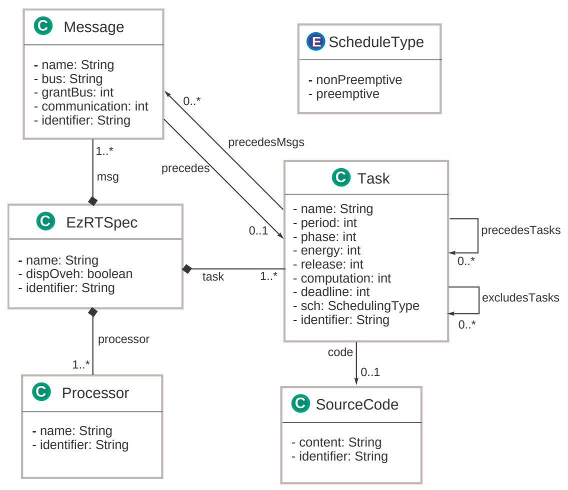

Figure 5: Overview of the implemented Specification metamodel

4.3 The tool architecture of ezRealtime

In effect, we created a DSML based on a time Petri net formalism and the *EMF.Codegen* code generator facility translates this model into a treeview graphical editor, where end-users define a set of **Tasks** and their inter-relations, whether such tasks are executed in one or more processors, 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 by following the proposed Building Blocks (Section 3.3.1), Operators and Net Compositions (Section 3.3.2) approaches, and it serves as base for the pre-runtime ezRealtime *Scheduler* engine that is used to find a feasible schedule, and then low-level C code is automatically synthesized.

The software is therefore broken into standalone, well defined tasks, and then the ezRealtime Scheduler engine produces a feasible schedule ensuring that timing, energy, precedence and exclusion constraints are satisfied for these tasks.

One of the goals of the project is to provide a straightforward mechanism for developing embedded hard real-time systems. Therefore, end-users do not necessarily know that there is an underlying formal semantics that provide the basis for the automation of software synthesis. The tool architecture of ezRealtime is illustrated in Figure 6.

This work considers that the synthesized system runs without a multitasking real-time operating system (RTOS), since it generates timely and predictable scheduled code.

It is important noticing that as we have shown, different perspectives exist when examining the internal and external aspects of ezRealtime. From the tool developers' viewpoint, the labor involved would be in: i) the up-front creation of an accurate metamodel to accomplish the domain-specific goals, ii) the adoption of the right technology and tool for developing the DSML, iii) the construction of the graphical editor, where the end-users will input the information about the hard real-time software under development, iv) the development of the code generator engine. From the end-users' viewpoint, the labor involved would be in: i) modeling all

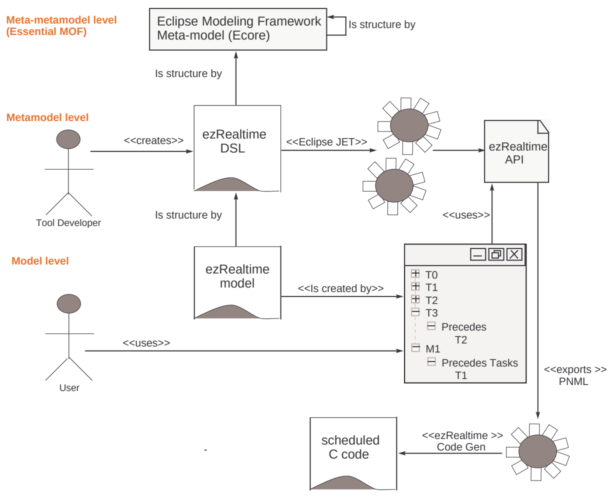

Figure 6: tool architecture of ezRealtime

relevant system's Tasks and their relations, ii) invoke the compiling feature provided by the C code generator engine.

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[4] for mapping from ezRealtime DSL (see Figure 7) into timed Petri nets through the PNML (ezRealtime2PNML).

The PNML Framework [7] is the API used in this work and it provides an extensive and comprehensive API to create and save PNML models, load and fetch PNML models elements. In this context, the PNML Framework will be used by the ezRealtime Exporter component to translate the DSL below into a PNML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<rt:ez-spec xmlns:rt="http://pnmp.sf.net/EZRealtime">
  <Task precedesTasks="#ez1151891690363" identifier="ez1151891">
    <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: a XML representation of the hard real-time software specification

4.4 Code Generator Engine

A ezRealtime *CodeGen* library was developed in order to automate the code generation process. Such engine uses the Ruby[5] programming language to write code for microcontrollers. This section shows the scheduler synthesis and the

[4]PNML Framework, http://www.lip6.fr/pnml
[5]http://www.ruby-lang.org

```
1  scheduling-synthesis(S,M^F,TPN)
2  {
3    if (S.M = M^F) return TRUE;
4    tag(S);
5    PT = pruning(firable(S));
6    if (|PT| = 0) return FALSE;
7    for each (⟨t,θ⟩ ∈ PT) {
8       S'= fire(S, t, θ);
9       if (untagged(S') ∧
10         scheduling-synthesis (S',M^F,TPN)){
11         add-in-trans-system (S,S',t,θ);
12         return TRUE;
13      }
14   }
15   return FALSE;
16 }
```

Figure 8: Scheduling Synthesis Algorithm

scheduled code generator.

4.4.1 Pre-Runtime Schedule Synthesis

The ezRealtime *Scheduler* component automatically looks for a feasible schedule. It takes the PNML document as input and if a schedule is found, the tool generates a scheduler to control the tasks' execution. It is worth observing that this is pre-runtime scheduling policy, which is fundamental to satisfy timing requirements established in the specification.

Starting from the time Petri net model, the proposed scheduling synthesis framework generates and analyzes the timed labeled transition system (TLTS), as a result of this model. It also analyzes the TLTS in order to find a pre-runtime schedule, provided that such schedule exists.

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 maintain the state space growth under control, the proposed method adopts a partial-order minimization technique [10] in order to prune the state space.

The proposed algorithm (Fig. 8) is a depth-first search method on a generated on-the-fly timed labeled transition system (TLTS). The *stop criterion* is obtained whenever the desirable final marking M^F is reached. Due to lack of space, the interested reader is referred to [2] for more details of the algorithm.

The only way the algorithm returns TRUE is when it reaches a desired final marking (M^F), implying that a feasible schedule was found (line 3). The state space generation algorithm is modified (line 5) to incorporate the state space pruning. PT is a set of ordered pairs $\langle t, \theta \rangle$ representing for each firable transition (post-pruning) all possible firing time in the firing domain. The *tagging scheme* (lines 4 and 9) ensures that no state is visited more than once. The function `fire` (line 8) returns a new generated state (S') due to the firing of transition t at time θ. The feasible schedule is represented by a timed labeled transition system that is generated by the function `add-in-trans-system` (line 11). When the system does not have a feasible schedule, the whole reduced state space is analyzed.

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. The timer interrupt handler always transfers the control to the dispatcher, which evaluates the need of performing either context saving or restoring, and after that to call the specific task. 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 9 depicts the schedule table for a preemptive application and Figure 10 presents the respective timing diagram. It includes two instances of `TaskA`, two instances of `TaskB`, two instances of `TaskC`, and one instance of `TaskD`. `TaskA1` and `TaskA2` are preempted in time 4 and 20, respectively. `TaskB1` is preempted twice: first in time 6 and, then, in time 10. Therefore, the schedule table contains 11 entries.

```
struct ScheduleItem scheduleTable [SCHEDULE_SIZE] =
{{ 1, false, 1, (int *)TaskA}, /* TaskA1 starts */
 { 4, false, 2, (int *)TaskB}, /* TaskB1 preempts TaskA1 */
 { 6, false, 3, (int *)TaskC}, /* TaskC1 preempts TaskB1 */
 { 8, true,  2, (int *)TaskB}, /* TaskB1 resumes execution */
 {10, false, 4, (int *)TaskD}, /* TaskD1 preempts TaskB1 */
 {11, true,  2, (int *)TaskB}, /* TaskB1 resumes execution */
 {13, true,  1, (int *)TaskA}, /* TaskA1 resumes execution */
 {18, false, 1, (int *)TaskA}, /* TaskA2 starts */
 {20, false, 3, (int *)TaskC}, /* TaskC2 preempts TaskA2 */
 {22, false, 2, (int *)TaskB}, /* TaskB2 starts */
 {28, true,  1, (int *)TaskA}  /* TaskA2 resumes execution */
};
```

Figure 9: Example of a Schedule Table

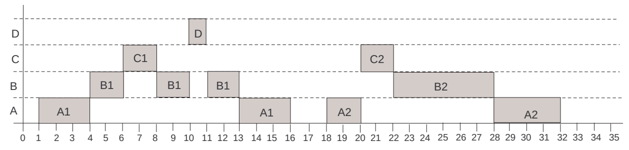

Figure 10: Timing Diagram for Schedule Table

5. CASE STUDY: MINE SYSTEM

This case study is a real-world application, where detailed specification for this example can be found in [4]. 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 presents the system specification. The system has five sensors: CH_4, CO, water flow, air flow, and high/low water. For each sensor one handler task is defined. The first four are periodic and the latter is sporadic. Another four

Table 1: Specification for Mine Drainage System

task	C	D	P
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

sporadic tasks are *pump motor control* (PMC), *water flow check* (WFC), *process data logger* (PDL), and *store data logger* (SDL). All sporadic tasks were translated into a periodic ones. The CH4S task is responsible for verifying the CH_4 safety level. The PMC task is used to represent the control over the motor switch. 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, using a non-preemptive (NP) method. 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 (EHRT) systems and for synthesizing timely and predictable scheduled C code, which can be leveraged in the applications. Such software uses its transformation component library for mapping the proposed DSL into the rigorous semantics of time Petri nets.

We are evolving the ezRealtime project, both to improve its CodGen component and transformation rules. Our aim is to apply the proposed methodology in the development of the EHRT software for all sorts of microcontrollers and processors (e.g., ARM9, 8051, M68K, x86) in a generative way. Moreover, the generated code should also be tuned and optimized to specific platforms.

This paper contributed with the scheduled code generator that receives the EHRT software specification and checks if there exists a feasible schedule for the system. If yes, the schedule is automatically generated and the code for the system is synthesized considering a pre-runtime scheduling strategy. The **ezRealtime** tool per si is a contribution. The more specific contributions to the DSL Engineering and EHRT domains are depicted as follows: (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 the development of predictable scheduled code for EHRT systems.

7. ACKNOWLEDGMENTS

The authors would like to thank the support received from the Nokia Institute of Technology (INdT) and the Brazilian agency CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) – Process number: 553164/2005-8.

8. ADDITIONAL AUTHORS

Additional author: Paulo Maciel (Centro de Informática (CIn), Universidade Federal de Pernambuco, Brazil, 50732-970 PO Box 7851 - email: `prmm@cin.ufpe.br`).

9. 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] O. M. Group. Meta-object facility 2.0 specification. Specification Version 2.0, Object Management Group, January 2006.

[7] L. Hillah, F.Kordon, L. Petrucci, and N. Trèves. Pn standardisation: a survey. In E. Najm, J.-F. Pradat-Peyre, and V. V. Donzeau-Gouge, editors, *FORTE'06*, volume LNCS 4229, pages 307–322. Springer, September 2006.

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

[9] 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, Budapest, Hungary*, volume 17, May 2001.

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

[11] 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.

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

[13] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[14] J. Sztipanovits and G. Karsai. Generative programming for embedded systems. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 180–180, New York, NY, USA, 2002. ACM Press.

[15] 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.