Automatic Code Generation from Real-Time Systems Specifications

L. Carnevali, D. D'Amico, L. Ridi, E. Vicario
Dipartimento di Sistemi e Informatica - Universitá di Firenze
carnevali@dsi.unifi.it, dario.damico@damix.it, ridi@dsi.unifi.it, vicario@dsi.unifi.it

Abstract

We address the problem of rapid development of complex real-time task-sets through a Model Driven Development (MDD) approach. The task-set is specified according to the graphic formalism of timeline schemas and it is translated into C-code that implements the dynamic architecture of the task-set on top of Linux-RTAI operating system. The transformation is performed through an engine obtained as an instance of a new model-transformation framework based on Java and eXtensible Stylesheet Language Transformations (XSLT) called JComposer. This is designed according to a flexible architecture that enables agile evolution of specification formalisms and target artifacts employed along the development process.

1. Introduction

The development of reactive and time-dependent systems faces complexities related to non-functional requirements, concerning both logical and timing correctness. To cope with these difficulties and reduce the development effort, certification standards [11][9] encourage the use of formal methods supporting validation at design stage, code-generation and testing activities [6][8][7]. This becomes crucial in the context of *Model Driven Development* (MDD) where specification models are systematically compiled into code and various other artifacts.

Various tools [17][1][14] have been described and experimented in the application of the MDD concept, with different goals in the aspects of model formalization, validation, implementation and verification. As a common trait, all these experiences give prominence to the automatic development of artifacts, which includes: the translation of a model that meets industrial acceptance into a formal model that supports automation of various activities such as analysis, code generation, test-case generation and selection of inputs for sensitization purposes, design documentation meeting certification prescriptions. In general, all these transformation processes can be described under a common abstraction [15][20] that identifies: *i) Domain Specific Modeling Languages* (DSMLs) for the formalization of the application structure, behavior, and requirements within a particular domain; *ii) transformation engines*, represented by a framework supporting the development of generators; *iii) generators* of *artifacts*, that analyze certain aspects of models and then synthesize various types of concrete artifacts, such as source code, simulation inputs, XML deployment descriptions, or alternative model representations. This suggests the opportunity of a framework that supports a general approach to the definition of artifacts generators and that guarantees a certain level of abstraction over the whole transformation process, thus permitting its application to a variety of input formalisms and output artifacts.

In this paper, we present a new MDD framework called *JComposer*, based on Java and *eXtensible Stylesheet Language Transformations* (XSLT), and we illustrate its application in the automatic generation of C-code for Linux-RTAI [18] from a visual specification (*timeline schema*) of the dynamic architecture of real-time task-sets. Two different artifacts are created and both the translations are supported within the Oris Tool [19][3] through JComposer: on the one hand, the timeline specification is translated into the formal model of *preemptive Time Petri Nets* (pTPNs) that make the specification amenable to simulation and state space analysis [2]; on the other hand, real-time C-code is derived which comprises an executable architecture of the task-set, with entry points enabling incremental integration of functional low level units. This comprises two relevant aspects: *i)* the translation process develops within a safety critical context and it is property-preserving with respect to the semantics of pTPNs, supported by a strategic theoretical corpus; *ii)* the SW architecture of JComposer enables the construction of extendable artifact generators, thus com-

prising a strategic hinge [10] that facilitates evolution and adaptation to changing needs in the context of use. In this paper, we mainly focus on the latter aspect.

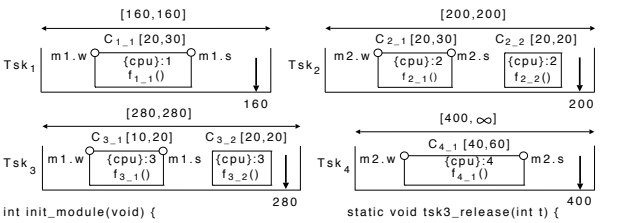
The rest of the paper is organized as follows. The implementation of a timeline specification on top of Linux-RTAI is reported in Sect.2. The SW architecture of JComposer and its application to source code generation are discussed in Sect.3. Finally, conclusions are drawn in Sect.4.

2. Implementing a timeline specification on top of Linux-RTAI

We address specification and implementation of complex real-time task-sets, comprised by recurrent tasks that release jobs with *periodic*, *sporadic* or *jittering* policy, depending on whether *i)* the release time is deterministic, *ii)* the release time has a minimum but not a maximum value, or *iii)* the release time is constrained within a minimum and a maximum value. Jobs are composed of sequential chunks, each having a non-deterministic execution time constrained within a minimum and a maximum value. Chunks pertaining to different jobs may be synchronized through semaphores, they may require resources according to a priority level and run under static-priority preemptive scheduling.

Timeline schemas are an intuitive graphic formalism widely employed in the practice of real-time systems [5][7] for the representation of a task-set architecture, enabling the annotation of tasks with relevant factors such as the deadline and the *entry points* (i.e. function names for the attachment of functional behavior of low-level units to chunks). Fig.1 reports the timeline schema of a task-set comprised of three periodic tasks (Tsk_1, Tsk_2, Tsk_3, with periods of 160, 200, 280 time units, respectively) and a sporadic task (Tsk_4, with minimum inter-arrival time of 400 time units), synchronized by two semaphores (m_1, m_2). Tsk_1 has a deadline of 160 time units and it is composed of a unique chunk C_{1_1} with entry point $f_{1\_1}()$, whose computation requires resource *cpu* with priority level 1 and lasts for a time constrained within 20 and 30 time units. C_{1_1} is synchronized with the first chunk of Tsk_3 through semaphore m_1.

Referring to the example of Fig.1, we recall here the salient traits of the disciplined coding that enables the implementation of a task-set on top of Linux-RTAI [7], providing an *executable architecture* [16] responsible for task releases, semaphore and priority handling operations, sequenced invocation of entry-points functions associated with chunks (the implementation of functional code is a duty of the programmer). Each task in the timeline specification is mapped on a recurrent real-time task in the implementation, performing recurrent release of jobs; a job has a counterpart in a one-shot real-time task, that invokes entry-point functions associated with chunks, wait and signal semaphore operations and priority handling operations (e.g. the real-time task `Tsk3` performs job releases of Tsk_3 and, according to this, the body loop of its function `tsk3_release()` spawns the real-time task `Tsk3Job`, which is associated with function `tsk3_job()` and performs a job of Tsk_3 assuming *priority ceiling emulation protocol* [21]). The task-set is implemented as a kernel module, whose function `init_module()` is responsible for creation of data structures (e.g. semaphores `m1` and `m2`), release of recurrent real-time tasks (e.g. `Tsk3`) and timer start. The executable architecture can be instrumented so as to support testing activities [7][8]. In particular, this permits to obtain a time-stamped log of events (i.e. the release of a job, the completion of a chunk, the completion of a semaphore wait operation, the completion of a priority boost operation) which can be off-line analyzed to emit a verdict about sequencing and timeliness conformance of executed runs.




```
int init_module(void) {
    rt_typed_sem_init(&m1, 1, BIN_SEM);
    rt_typed_sem_init(&m2, 1, BIN_SEM);
    ...
    int tsk3Prio = 0;
    long long tsk3Period = 280000000;
    rt_task_init(&Tsk3, tsk3_release, 0,
                 2000, tsk3Prio, 0, 0);
    ...
    rt_set_periodic_mode();
    start_rt_timer(nano2count(500000));
    long long startTime = rt_get_time_ns();
    ...
    long long tsk3Start = startTime + tsk3Period;
    ...
    rt_task_make_periodic(&Tsk3,
                nano2count(tsk3Start),
                    nano2count(tsk3Period));
    return 0;
}
```

```
static void tsk3_release(int t) {
    int tsk3JobPrio = 3;
    while(1) {
        rt_task_init(&Tsk3Job, tsk3_job,
            0, 2000, tsk3JobPrio, 0, 0);
        rt_task_resume(&Tsk3Job);
        rt_task_wait_period();
    }
}
static void tsk3_job(int t) {
    rt_change_prio(rt_whoami(), 1);
    rt_sem_wait(& m1);
    // Computation of the first chunk
    f3_1();
    rt_sem_signal(& m2);
    rt_change_prio(rt_whoami(), 3);
    // Computation of the secon chunk
    f3_2();
    rt_task_delete(rt_whoami());
}
```

Figure 1. The timeline schema and fragments of its implementation.

3. JComposer and RT code generation

XSLT [22] is a declarative language for the transformation of XML documents and can be employed in the context of MDD to derive output documents with any format. An *XSLT processor* is a SW object that builds a *source tree* from the input XML document and

processes its nodes according to template rules defined within an XSLT stylesheet (XSLT elements are defined in the namespace `xsl`).

XSLT supports the manipulation of structural aspects of a system model but it is not suitable for dealing with behavioral features, that can be natively handled by imperative programming languages (e.g. Java) through the implementation of algorithms and procedures. With reference to the code generation issue discussed in this paper, it is worth noting that standard XSLT elements enable the definition of rules according to which structural components of the timeline specification are mapped into portions of code (e.g. each task in the specification corresponds to a recurrent real-time task in the implementation), but they are not suited to express transformation rules that involve behavioral features of the model (e.g. boost and deboost operations according to a priority protocol). To overcome the limitation, the *JComposer* framework defines a new language called *JComposer XSLT* (*jcXSLT*), that extends syntax and semantics of XSLT with additional elements defined in a separate namespace `jc`. Processors are distinct in *Java processors* and *jcXSLT processors*: the former manipulate the input document without relying on an external stylesheet of template rules; the latter operate the transformation according to the rules defined within a jcXSLT stylesheet, which combines `xsl` and `jc` elements. `jc:exec-java`, `jc:immediate` and `jc:forecast` elements allow a jcXSLT processor to interact with Java processors.

A `jc:exec-java` element specifies the name of the Java class implementing a Java processor, and its semantics instructs a jcXSLT processor to instantiate and invoke the specified Java processor on the content of the `jc:exec-java` element, which is replaced by the result returned by the Java processor. `jc:exec-java` elements are processed after XSLT elements, unless they are nested within a `jc:immediate` element, which forces a jcXSLT processor to execute its content before any other tag. In case a `jc:immediate` element contains a `jc:forecast` tag, XSLT elements nested within the `jc:forecast` tag are processed before any `jc:exec-java` element contained within the `jc:immediate` tag. This enables a Java processor to receive an input document calculated through XSLT rules. If two or more `jc` elements of the same type (`jc:exec-java`, `jc:immediate`, `jc:forecast`) are nested, they are processed starting from the most internal one.

The JComposer framework has been exploited in the construction of various transformation engines in the Oris Tool [19][3], each comprised by a set of processors. In particular, we briefly describe the engine that converts timeline specifications into real-time code. It provides a graphical interface for the specification of user preferences, which mainly account for the selection of a priority protocol (by now, only the priority ceiling emulation protocol [21] is supported) and code instrumentation options for testing purposes (e.g. time-stamped log of job releases and completions). The transformation process spans various faces, during which an XML file encoding the timeline specification is progressively enhanced with information coming from user options (e.g. the selected priority protocol is stored as an attribute of the XML element representing the whole timeline) and it is finally mapped into source code. As a relevant example, we report below the jcXSLT rules that instruct the jcXSLT Processor `JcProc` to determine whether a chunk requires a priority boost. The `xsl:for-each` element specifies a `for` loop that iterates over all chunks, and at each repetition: *i)* `JcProc` executes XSLT rules contained within the `jc:forecast` element, selecting the id of a chunk, the id of the task it belongs to and the XML code of the whole timeline; *ii)* `JcProc` instantiates and invokes the Java processor `PriorityProc` passing the information derived at the previous step as input document; *iii)* `PriorityProc` determines whether the chunk requires a priority boost, calculates boosted priority level if needed and returns this information stored within an XML element; *iv)* `JcProc` replaces the `jc:immediate` tag and its nested tags with the returned XML element. `PriorityProc` is designed according to the *Strategy* pattern [12], with each priority protocol implemented as a *Concrete Strategy*.

```
<xsl:for-each select="tl:chunks/tl:chunk">
 ...
 <xsl:variable name="priorityChange">
  <jc:immediate>
   <jc:forecast>
    <jc:exec-java proc="processors.PriorityProc">
     <priority-action-input>
      <task-id>
       <xsl:value-of select="$tsk/@id"/>
      </task-id>
      <chunk-id>
       <xsl:value-of select="$chk/@id"/>
      </chunk-id>
      <taskSet>
       <xsl:copy-of select="/tl:tlml"/>
      </taskSet>
     </priority-action-input>
    </jc:exec-java>
   </jc:forecast>
  </jc:immediate>
 </xsl:variable>
 ...
</xsl:for-each>
```

4. Conclusions

The main contribution of this paper consists in the definition of the MDD framework JComposer and in its application to real-time code generation. JComposer is based on Java and XSLT and supports the construction of extendable artifacts generators, thus comprising a strategic design *hinge* [10] in the adaptation to various needs and demands.

The JComposer framework has been employed in the automatic generation of real-time C-code for safety-critical embedded systems run on top of Linux-RTAI platform. As a characterizing trait, the resulting transformation engine exhibits a flexible architecture with respect to various evolutions:

- system design is described according to the formalism of timeline schemas but it would equivalently be specified through other paradigms (e.g. notably, the UML profile for MARTE or AADL with suitable restrictions/extensions and/or pragmatic interpretation) and the adaptation can be easily accomplished by implementing the transformation process that converts a system model specified through the selected formalism into the corresponding timeline;
- changing the target language (e.g. Ada, C++), the running space (e.g. user mode instead of kernel mode), the RTOS (e.g. RT-Linux, VxWorks) or code constraints (e.g. programming under the Ravenscar profile [4], MISRA C programming [13]) requires the adaptation of some jcXSLT rules without any maintenance of the existing Java code;
- adding the support for a new priority protocol can be achieved by implementing a new Concrete Strategy [12] for the Java processor `PriorityProc` that handles transformation steps pertaining to the selected priority protocol.

References

[1] R. Alur, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Hybrid Systems: Computation and Control, LNCS 2034*, pages 33–48. Springer–Verlag, 2001.

[2] G. Bucci, A. Fedeli, L. Sassoli, and E. Vicario. Timed state space analysis of real time preemptive systems. *IEEE Trans. on Soft. Eng.*, 30(2):97–111, Feb. 2004.

[3] G. Bucci, L. Sassoli, and E. Vicario. Oris: a tool for state space analysis of real-time preemptive systems. *Proc. of the* 1^{st} *Int. Conf. on the Quant. Evaluation of Sys. (QEST)*, September 2004.

[4] A. Burns, B. Dobbing, and T. Vardanega. Guide on the use of the ada ravenscar profile in high integrity systems. *ADA Letters*, XXIV(2):1–74, 2004.

[5] G. Buttazzo. *Hard Real-Time Computing Systems*. Springer, 2005.

[6] L. Carnevali, L. Grassi, and E. Vicario. A tailored v-model exploiting the theory of preemptive time petri nets. In *Ada-Europe '08: Proc. of the Ada-Europe Int. Conf. on Reliable Software Technologies*, pages 87–100, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] L. Carnevali, L. Sassoli, and E. Vicario. Casting preemptive time petri nets in the development life cycle of real-time software. *Proc. of the Euromicro Conference on Real-Time Systems*, July 2007.

[8] L. Carnevali, L. Sassoli, and E. Vicario. Sensitization of symbolic runs in real-time testing using the oris tool. *Proc. of the IEEE Conf. on Emerging Technologies and Factory Automation (ETFA)*, Sept. 2007.

[9] CENELEC. En 50128 - railway applications: Sw for railway control and protection systems. Technical report, CENELEC, 1997.

[10] M. P. Cline. The pros and cons of adopting and applying design patterns in the real world. *Commun. ACM*, 30(10):47–49, 1996.

[11] R. T. C. for Aeronautics. Do-178b, software considerations in airborne systems and equipment certification. Technical report, RTCA, 1992.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[13] L. Hatton. Language subsetting in an industrial context: a comparison of MISRA C 1998 and MISRA C 2004. *Information and Soft. Tech.*, 49(5):475–482, 2007.

[14] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proc. of the IEEE*, pages 84–99. IEEE, 2003.

[15] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proc. of the IEEE*, 91:145–164, Jan. 2003.

[16] P. Kruchten. *The Rational Unified Process: an introduction*. Addison–Wesley, 2003.

[17] The Mathworks. *Simulink*. http://www.mathworks.com/products/simulink/.

[18] D. of Aerospace Eng.of the Polytechnic of Milan. *RTAI: Real Time Application Interface for Linux*. https://www.rtai.org.

[19] L. Sassoli and E. Vicario. Analysis of real time systems through the oris tool. *Proc. of the* 3^{rd} *Int. Conf. on the Quant. Evaluation of Sys.(QEST)*, Sept., 2006.

[20] D. C. Schmidt. Model–driven engineering. *IEEE Computer*, pages 1–2, February 2006.

[21] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[22] World Wide Web Consortium. *XSL Transformation (XSLT)*, November 1999. http://www.w3.org/Style/XSL/.