Automatic generation of controls code from models for real-time Linux platforms

Bruno Morelli, Riccardo Schiavi, Claudio Scordino, Paolo Gai
Evidence Srl
Pisa, Italy
{b.morelli, r.schiavi, claudio, pj}@evidence.eu.com

Marco Di Natale
Scuola Superiore Sant'Anna
Pisa, Italy
marco@sssup.it

Abstract

We present our experience in the design and realization of a toolset for the extension of the simulation and modeling capabilities and the automatic generation of code from ScicosLab models running on embedded Linux and RTAI. ScicosLab — a free system modeler and simulator — has been extended with add-ons for the modeling of hierarchical Finite State Machines (FSMs) and a GUI prototyper based on the Qt graphical libraries. The code generator supports multirate models and maps the model blocks onto multiple periodic threads, with the automatic synthesis of thread scheduling, communication and access to the I/O primitives.

We discuss the design challenges and options and the implementation issues encountered during the development of the toolset, including the mapping of functionality onto threads, the problem of semantics preservation in the implementation of communication, the methods used to abstract from the architecture-specific I/O details and the concurrent use of PCI and video peripherals. We show an example of generated code and a set of performance measurements on real hardware with a data acquisition peripheral.

1 Introduction

The Model-Based Design (MBD) development methodology is today widely used in several application domains like automotive, aerospace and controls. MBD can provide a significant reduction of development time and costs for embedded software by leveraging early verification and validation.

In MBD, a model of the control functions is developed together with a model of the controlled system (or plant). The behavior (within and possibly also outside the specification range) is verified by simulation and — in safety critical systems — by model checking. Automatic code generation techniques are then used to generate a software implementation that is equivalent to the model, reducing the possibility of adding errors during the manual coding stage.

Several commercial products are offered for the modeling, simulation and verification of systems according to a Synchronous Reactive (SR) semantics. Among them, Simulink, LabView and SCADE are probably the best known. ScicosLab [6], a free system modeler and simulator developed at Inria, starting from the late '90s is a free and open source alternative for system simulation. However, the use of ScicosLab for the industrial development of embedded controls has been limited due to a number of issues, including the lack of support for modeling finite state machines, a code generator not suitable for systems with limited memory availability, no

support for multithreaded implementations (of multirate models), and platform-specific code, including the use of I/O. We present our experience in the design and realization of a toolset that attempts at filling these gaps. Our toolset consists of

- A tool for the modeling, simulation, and automatic software synthesis of extended Finite State Machines, (FMS), called **SMCube**. SMCube supports hierarchical and concurrent FSMs and provides a simple graphical interface for the modeling and simulation, including animation and tracing.
- **E4CoderGUI**, a GUI prototyper based on the Qt graphical libraries [7], with a graphical WYSIWYG editor, a simulator and a runtime generator for Linux platforms.
- **E4CoderCG**, a code generation framework based on model-to-model and model-to-code transformation techniques, with support for datatypes, efficient code inlining and generation of a mutithreaded implementation (OSEK, Linux or RTAI) of multirate models.
- Support for I/O devices in the model and at code generation time through the use of a hardware abstraction layer. The integration with I/O and data acquisition in Linux and RTAI is provided by custom blocksets based on the Comedi libraries [3].

The paper is organized as follows. In Section 2, we provide an outline of related work and tools. In Section 3, we describe the decisions made during the design and development of our toolset. In Section 4, we explain how code is generated on both native Linux and RTAI systems and show an example of generated code. In Section 5, we describe the issues and challenges that we encountered and provide a set of performance measurements on a platform with a data acquisition peripheral. Finally, in Section 6, we state our conclusions and future work.

2 Related work

Despite the rise of multi-core architectures in the embedded domain, commercial MBD tools offer an implementation path that is mostly oriented to single-core platforms, with little or no support for thread allocation and very limited support for the definition of concurrent thread implementations and the use of OS, communication, and device driver primitives. Free or Open-Source tools for MBD have been historically limited by the lack of efficient code generation methods and the absence of integrated support for dataflows and finite state machines.

Model-Based Development leverages formal (graphical) modeling languages for the high-level representation of control functionality according to a synchronous reactive semantics. Examples of such tools are Simulink from Mathworks and SCADE from Esterel. SCADE provides graphical editing capabilities to the Lustre and Esterel Languages [19] and the SyncCharts extension [20]. These languages have been developed from research labs and, at some point in their development, were available (possibly in limited form) as free tools.

Today, the most promising free alternative for the modeling and simulation of systems is ScicosLab. Based on the Metalau project, ScicosLab offers a TCL-based graphical front-end, a simulation engine and a code generator for the purpose of simulation on the host. The idea of extending the MBD capability of ScicosLab with support for embedded targets and open-source RTOS is not new. The RTAI-Lab toolbox, based on the code generation developed by Bucher [10] allows to generate executable code starting from a ScicosLab superblock into a single periodic task in RTAI. The code can then be connected to external tools (XRTAILab, QRTAILab [14], RTAI-XML [15]) to display data remotely. The code generator has been extended to support an OSEK/VDX operating system (such as ERIKA Enterprise [11, 13]).

However, the existing code generator has several limitations. First and foremost, each atomic block translates into a single function call, which leads to large code size and overheads that may be unacceptable for small systems. Then, to reuse the simulator features, data typing is often limited and most types are by default mapped to double precision. Finally, the only possible code generation option is for a single-rate single task implementation and I/O access (or any platform-specific code) is supported through platform-specific custom blocks or requires code stitching by hand.

In addition, ScicosLab does not support modeling of finite state systems and subsystems. This is a significant limitation for most commercial controls that are heavily state-dependent and based on a switching logic. The motivation for this omission is the original emphasis on physical system modeling and simulation. The fix, however, is not simple. Finite state machines do not come in a single flavor, and even less when it comes to the need to define a graphical language (in addition to the execution semantics). Besides the SyncCharts proposal, other examples of synchronous state machines are StateFlow by Mathworks [21] and the proposed addition to the Modelica Language [22]. Asynchronous state

machines also have a long history. The original Statecharts proposal by Harel [18] was implemented by Ilogix in the Rhapsody tool (now IBM). The OMG provided its own version in the State Diagrams, as part of the UML standard [23].

Code generation from models is the subject of many other projects. Gene-Auto is an open-source toolset for converting Simulink, Stateflow and Scicos models to executable program code. Project P and Hi-MoCo [17] (High-Integrity Model Compiler) are two open-source research efforts supported by the French and Estonian national governments and the European EUREKA agency, to provide an open-source, tunable and qualifiable code generator for domain-specific modeling languages.

3 System design

The SMCube and E4CoderGUI tools consist of two modules: a component that is executed at simulation time, and a component for the generation of code to be executed on the target.

The components executed at simulation-time are connected to the ScicosLab simulation engine through the standard mechanism for the definition of custom blocks. A custom block is defined in the ScicosLab model for each state machine or GUI subsystem. The custom block connects to the other blocks in the ScicosLab model through standard input and output ports, and opens local sockets for communication with an external program providing the simulation of the FSM subsystem or GUI interface, respectively. The simulation executes synchronously in the ScicosLab part with the external programs.

Figure 1 shows the workflow for the generation of code. The code for each SMCube and E4CoderGUI subsystem is generated by a dedicated engine that processes the model information (encoded in XML) and generates a single `Step` function that computes at runtime the state and output update functions of the subsystems. The code generated for the ScicosLab model follows a different path. Three different modules support the generation of the *functional code* (implementing the behavior of the dataflow part of the model); the *thread setup and structure*; and the *I/O part*. The first translation is performed through model-to-model and then model-to-text transformations. The thread creation model has a dedicated engine and the I/O access is performed through a simple abstraction layer that allows a view of I/O peripherals in the model independent from the platform. A separate generator provides the configuration of the abstraction layer based on hardware-specific information.

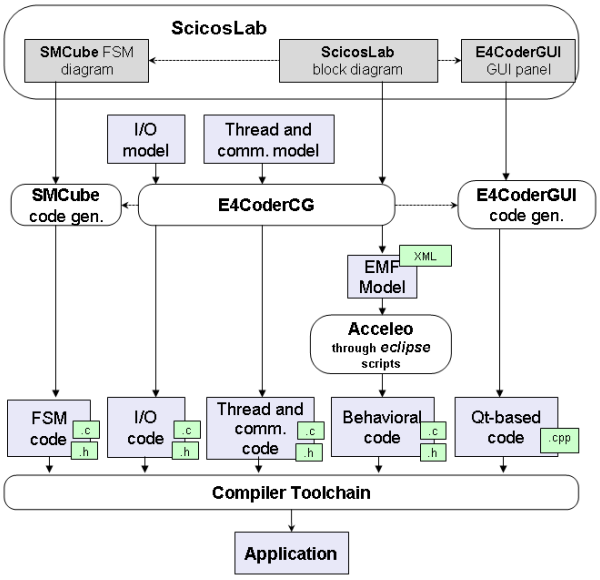

FIGURE 1: *Workflow of code generation.*

3.1 E4CoderCG

E4CoderCG generates the C code implementation of the ScicosLab model. Code generation can be performed for the entire model or for a selected subset (possibly one) of its subsystems. The code generator operates in steps. The model is exported (through an intermediate XML format) to Eclipse EMF as an Ecore model defined according to our proprietary metamodel. The Ecore model is then processed by a model-to-text transformation (a C code generator) defined using the open tool Acceleo [9]. Decoupling the code generation through an intermediate model allows for more flexibility and control in the generation stage and opens the door for code generation from other languages/tools as well as a tighter integration with other Eclipse (modeling) tools.

The model-based design of a control system typically consists of an initial simulation phase, in which the control logic is verified against the model of the system to be controlled (or plant, typically as a set of differential equations). During this initial phase, the hardware platform may be not available (e.g., simply because the final choice has not yet been made) and is virtualized or replaced by a rapid prototyping environment. Hence, our *peripheral blocks* are abstract and generic. An additional input (or output) port allows them to act as a passthrough (possibly with delay) at simulation time. Only at code generation time, each generic peripheral is mapped to the actual hardware I/O. In this way, the same diagram can be used for different hardware targets without

modifications. Moreover, the same diagram can be used for both simulation and code generation.

In the generation of a *multithreaded implementation* of a multirate model, the system must be composed of a set of first-level superblocks (subsystems), each activated by a single clock block (each superblock is single rate). Superblocks exchange signals through their ports. Connected to each port, at the boundary of the subsystem, there is a read/write block (communication block) or a generic peripheral block. Further, the synchronous execution semantics and the input/output dependencies dictate a partial order of execution among superblocks.

During the code generation, each first-level superblock is generated as a single C function. A wizard allows the user to define the set of periodic tasks (threads) in the implementation and to map the superblock functions for execution by a given task. The body of each task is periodically activated and executed by the underlying operating system (see Section 4). The mapping of blocks to tasks is allowed only if it is consistent with the execution rate and the order of execution of the blocks.

As for communication, when two first-level superblocks exchange data using data ports, the code generator tags each input/output port using read/write blocks (inserting them automatically, if not present). Then, the read/write blocks are implemented using a memory buffer protected with mutexes or interrupt-disabling, depending on the target system. Future enhancements will include communication using wait-free resources, fieldbus and other communication buses allowing the deployment of the generated code in distributed systems.

3.2 SMCube

SMCube is a modeler of hierarchical and concurrent finite state machines allowing interactive simulation and code generation in C. SMCube offers an editor and a simulation view. A snapshot of the simulator editor is shown in Figure 2.

To improve the performance, a C++ implementation of the FSM model is generated, compiled and executed at simulation-time in synchrony with the ScicosLab simulation. The code for the target is C code for minimum footprint.

Special care has been taken to reduce the number of possible semantics pitfalls and ensure that the generated code has the same behavior as the simulated model. Several modeling features and restrictions are provided and enforced. An execution order must be assigned to all transitions and parallel states, to guarantee a deterministic execution. In addition, the actions associated with transitions cannot produce events that trigger other transitions of the same machine (therefore guaranteeing a run-to-completion semantics, as opposed, for example to Stateflow). Also, recursive reactions are not allowed and side effects are only possible if the user chooses to provide transition actions as free code.

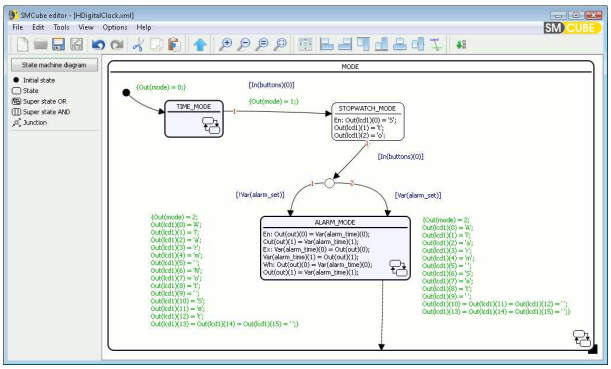

FIGURE 2: *SMCube editor for FSMs.*

3.3 E4CoderGUI

E4CoderGUI is a tool for the design and simulation of graphical user interfaces (GUIs). The designer part of E4CoderGUI is a standard WYSIWYG editor allowing the use of text, images and a number of other widgets and associating a set of predefined actions to the values of inputs and outputs (or conditions on them).

When inserted in a ScicosLab diagram, an E4CoderGUI block can be simulated providing means for the user to interact with the simulation, analyze results or simulate the operations of a touch screen or physical front-end. All inputs to the GUI are available to the model blocks through output ports. E4CoderGUI can be used for the simulation and fast prototyping of simple user interfaces (e.g., control systems, home appliances, HVAC controllers, automotive dashboards). These systems are typically composed by state machines and LCD displays with buttons. The tool allows users to test the behavior of the system before its deployment.

E4CoderGUI blocks can also be used during code generation. In those cases, the result is a Qt application showing the same panel constructed with the editor and validated during simulation. The generated code can be used in platforms and applications where the real-time controls and the HMI reside on the same executable (e.g., an embedded control Linux device with an LCD display).

Special care must be taken when generating code

for real-time systems because of the risk of jeopardizing the real-time performance of the control part when running the generated GUI code. In the next sections we provide examples of issues and solutions.

4 Generated code

The E4Coder generator provides support for several target operating systems and middlewares through a common execution model, consisting of a set of abstractions that must be implemented by each supported platform. Although each OS is characterized by its own peculiarities, we only require the implementation of a limited number of basic mechanisms, as required by our execution model. The coder supports the integration between the real-time subsystem (typically, control algorithms and mathematical functions) and the non-realtime subsystem (typically, the HMI as well as filesystem/network components for logging and calibration). Our execution model consists of the following concepts:

A **task** as a sequence of function calls derived from the execution of first-level superblocks in a given order. Each task needs to be activated according to its **period** (consistent with the sampling rate of the first-level superblocks mapped onto it).

A **priority** assigned to each task, and used to guarantee the correct order of execution of superblocks and to ensure schedulability.

The **background time**, or the time left free by periodic tasks, is typically used to perform background activities such as GUI updates.

The **initialization** and **end** of the tasks includes the initialization of device drivers to be performed with the right timing.

The runtime needs to guarantee the **consistency of the shared data**, implementing signals exchanged between superblocks mapped to different tasks [16].

The current version of E4Coder generates code for the following targets:

- Linux with pthread library;
- RTAI;
- OSEK/VDX - ERIKA Enterprise [11, 12];
- Bare metal;
- Windows (non-realtime).

The code generated for the Linux and the RTAI systems is integrated with the Comedi [3] libraries to allow easy access to most data acquisition boards.

4.1 Linux with pthreads

The main design abstraction is implemented as follows:

Each **task** is implemented using a pthread.

The **priority** of the task is mapped to the real-time priority of the pthread.

The **periodicity** is implemented by calling the superblock functions inside an endless loop including a call to `clock_nanosleep()`. A common start time is obtained by creating all pthreads beforehand, and letting them synchronize on a barrier just before and just after the initial timestamp is taken. Afterwards, before calling the first instance, a call to `clock_nanosleep()` waits for an offset when specified.

The **background time** is collected by the `main()` function. In case of the presence of an E4CoderGUI block, this function implements the Qt loop.

The **initialization** of each superblock is done before reaching the initial barrier. The **end** is caught either by a termination signal or by the closure of the Qt window, and it is implemented by calling the ending functions in each thread after exiting the endless loop.

Data sharing among tasks in a multirate system is implemented by using a priority inheritance mutex to protect from concurrent data access.

4.2 RTAI

The implementation on RTAI is similar to the one on Linux with the exception of the communication with the GUI. The communication between a real-time periodic task and the main Qt loop is done through Qt slots. However, the Qt code may call normal Linux primitives, leading to an automatic transition of the task from the RTAI real-time mode to the normal Linux mode. To avoid this risk, the code generator adds one additional "bridge" thread for each E4CoderGUI block (see Figure 3).

The bridge thread is responsible for the communication between the real-time task and the Qt framework. The communication with the real-time task is provided by a circular buffer that is not blocking for write operations (i.e., data is overwritten in case of buffer full). Thus, the real-time performance of the periodic task is not affected, while the E4CoderGUI panels execute in background using Qt.

When possible, the bridge thread is executed in the RTAI real-time user-space mode at the lowest priority. However, when calling Qt signals/slots it may enter normal Linux mode. In this case, it is executed at the maximum (i.e., `SCHED_FIFO`) priority as implemented by the following code:

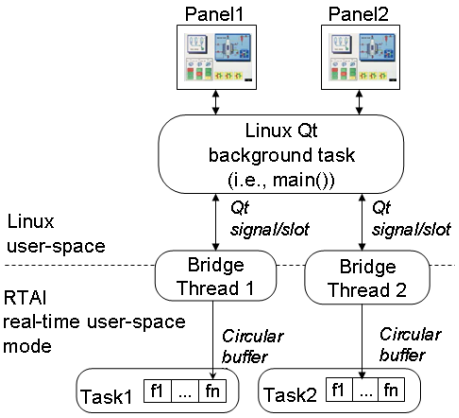

FIGURE 3: *Code generated for graphical applications on RTAI.*

```
void *GUI_io_gui_th(void *args)
{
    RT_TASK *rt_task;
    th_param_t *par = (th_param_t*)args;
    int in_elem = 100;
    int elem, i;
    gui_gui_data_in_t in[in_elem];
    rt_task = rt_task_init_schmod(nam2num(par->name),
        par->prio, 16 * 1024, 0, SCHED_FIFO,
        pthread_sync_param.rt_cpu_mask);
    rt_make_hard_real_time();
    par->queue_in  = e4c_rt_queue_init(par->qi_name, in_elem,
        sizeof(gui_gui_data_in_t));
    e4c_rt_barrier_wait(&pthread_sync_param.main_barrier);
    while (!par->end) {
        e4c_rt_queue_wait_data(par->queue_in);
        elem = e4c_rt_queue_get_all(par->queue_in, in);
        e4c_rt_queue_unlock(par->queue_in);
        rt_make_soft_real_time();
        for (i = 0; i < elem; ++i)
            GUI_io_gui(par->handle, in[i].input_0);
        rt_make_hard_real_time();
    }
    rt_make_soft_real_time();
    rt_task_delete(rt_task);
    return NULL;
}
```

4.3 OSEK/VDX

When dealing with OSEK/VDX RTOSs like ERIKA Enterprise [12], the amount of flexibility and available primitives is limited. Moreover, the typical target hardware is a microcontroller with limited resources. For this reason, not all the runtime supports available on Linux are also available on OSEK/VDX systems. However, despite the limitation of the API, all the main mechanisms requested to make a running control system are available.

Each **task** is simply implemented using an OSEK/VDX task.

The **priority** of the task has been mapped to the priority of the OSEK/VDX task.

The **periodicity** and **offsets** have been implemented by configuring an ALARM inside the OIL file. The ALARM is then linked to a COUNTER which is periodically activated by a timer. Periodicity is expressed as multiples of the timer frequency.

The **initialization** of each superblock is done in a dedicated task activated at system startup before starting all the periodic activities. The **end** part is not implemented since the typical microcontroller execution never ends.

Data sharing is currently implemented by simply disabling the interrupts, because the size of the data and the time needed to read/write them is negligible compared to the execution of an RTOS primitive.

Background activities and GUIs are not handled on OSEK/VDX systems.

4.4 Bare metal

A "bare metal" system is typically composed by a simple main loop executing function calls in sequence. This is a simple way to implement a non-preemptive real-time system without most of the overhead added by a real-time operating system. This approach works well for tiny systems with limited functionalities.

Each **task** is implemented as an `if` clause in the `main()` loop.

In this case there is no schedulability, and the **priority** of the task is simply the order of the `if` clause inside the main loop (each `if` clause terminates with a `continue` statement).

The **periodicity** and **offsets** have been implemented using integers incremented by a timer interrupt. Each task maintains its internal reference of time allowing each task to catch up in case it is not executed for a long time.

The **initialization** of each superblock is done in the `main()` function before entering the main loop. The **end** part is not implemented since the typical microcontroller execution never ends.

Data sharing is currently implemented by simply copying data (note that the system is non-preemptive).

Typically, **background activities** and GUI items are not handled on this kind of systems.

5 Experimental results

The efficiency of the code generated by our tool has been measured using the Atom™ D2550-based (dual-core 1.86 GHz) E4Box [2] platform with a National Instruments PCI-6221 data acquisition pe-

ripheral [5]. The distribution is Ubuntu with Linux kernel 2.6.38.8 and RTAI 3.9.

5.1 Issues

The first issue encountered during our experiments is related to RTAI not properly setting the task affinity. This issue has been fixed through a patch submitted to the RTAI community [4].

The second issue concerns the contention on some hardware component used concurrently by the graphic stack and our real-time application. The issue showed up when generating an output square wave, also drawn on the display, using the model in Figure 4. Our hypothesis is that the problem was related to the integrated graphic controller mounted on the PCI bus. Unfortunately, the Linux PCI stack does not take into consideration the priorities of the running tasks and requests are served in FIFO order, regardless of real-time priorities. In case of an extensive use of PCI peripherals, it could be worth investigating a patch for the Linux kernel to create time slots for the exclusive execution of real-time requests.

Finally, we discovered an issue related to the scheduling of infinite tasks on RTAI. Even on multi-core platforms, a single endless RTAI loop can freeze the whole CPU. This issue has been immediately reported to the RTAI community [4].

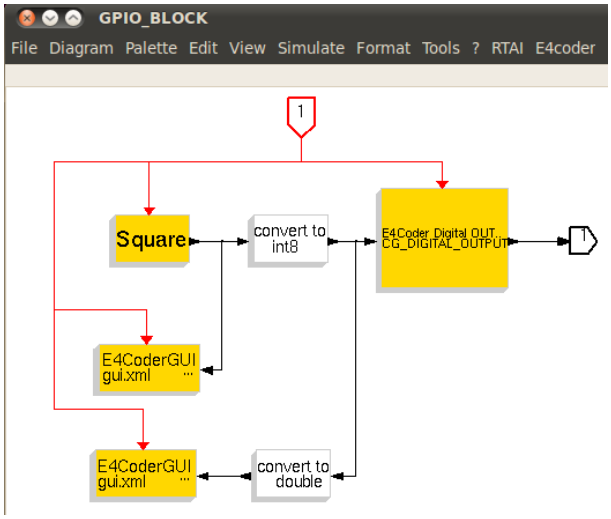

FIGURE 4: *Model for output square wave.*

5.2 Footprint analysis

The first experiment to measure the efficiency of our tool consists in the generation of a PID controller for a DC motor. The code (including the peripheral access) has been generated for the ERIKA Enterprise RTOS as a single task using Scicos-FLEX [13] (an open-source code generator derived from the Linux/RTAI code generator for ScicosLab) and our tool. The following table shows the footprint comparison in bytes.

	Scicos-FLEX	E4Coder
Total flash size	12393	8190
Total data size	2580	198
Control loop flash size	428	294
Control loop flash size with init	1998	304

TABLE 1: *PID controller footprint (bytes).*

Our code generator allows to save about 34% of the total footprint. In particular, the reduction of the total data size is 92% and 85% in the control loop.

5.3 Jitter analysis

To evaluate the performance of our real-time solution, we measured the latency of the square wave generated by the model shown in Figure 4. The latency has been measured by connecting an oscilloscope to the PCI-6221 board and registering the latency values. In particular, we measured the highest experienced latency on an idle system in several configurations:

- RTAI
- RTAI with cpu isolation, assigning Linux and graphics to one core and the real-time application to the other core (through the `isolcpu` and `noirqbalance` options).
- PREEMPT_RT on a Linux kernel 3.0
- Standard Linux

The experiments have been carried out with and without the GUI showing the wave on the display.

In the first experiment, the square wave had a 2msec period and a duty cycle equal to 50%. The longest latency measured in each configuration is shown in Table 2. Remember that in case of standard Linux, the real-time application is run with the `SCHED_FIFO` policy.

From the results, the use of cpu isolation on multi-core platforms is a valid mechanism to significantly reduce jitter. Similarly, the PREEMPT_RT patch allows to halve the latency in systems where RTAI is not available.

	Without GUI	With GUI
RTAI	10 μsec	70 μsec
RTAI isol.	10 μsec	40 μsec
PREEMPT_RT	20 μsec	140 μsec
Linux	40 μsec	260 μsec

TABLE 2: *Jitter for a 2msec wave.*

We repeated the same experiment by generating a square wave with period $200\mu sec$. Unfortunately, this experiment cannot be meaningful for PREEMPT_RT and Linux, due to excessive jitter, comparable with the wave period. The values measured for RTAI have been reported in Table 3. Again, cpu isolation allowed to obtain a sensible reduction of the experienced latency.

	Without GUI	With GUI
RTAI	4 μsec	20 μsec
RTAI isol.	2 μsec	7 μsec

TABLE 3: *Jitter for a 200μsec wave.*

6 Conclusions

In this paper we described the design decisions, the challenges and issues that we have encountered during the design and development of a toolset based on ScicosLab [6]. We discussed challenges and solutions for code generation and the target runtime system on several open-source operating systems. Finally, we provided experimental results obtained by running the generated code on Linux and RTAI on a real hardware equipped with a professional data acquisition peripheral.

As a future work, we plan to add better support for parameters of generic peripherals; support for other Open-Source RTOSs like Xenomai, FreeRTOS and RTEMS; support for simulation and debugging on SMCube; higher amount of available widgets on E4CoderGUI.

7 Acknowledgements

The authors would like to thank Giuseppe Arturi, Sara Corfini and Dario Di Stefano for their precious help in the design and development of several components of the E4Coder toolset [1].

References

[1] E4Coder, the toolset for simulation and code generation for embedded devices *http://www.e4coder.com*

[2] E4Box, Rapid Prototyping Hardware Box *http://www.e4coder.com/e4box*

[3] Comedi, Linux control and measurement device interface *http://www.comedi.org*

[4] RTAI, Single task 100% CPU stops a multicore machine *http://mail.rtai.org/pipermail/rtai/2013-June/025597.html*

[5] National Instruments NI PCI-6221 *http://sine.ni.com/nips/cds/view/p/nid/14132*

[6] ScicosLab, *http://www.scicoslab.org*

[7] Qt libraries, *http://qt-project.org*

[8] Syndex, *http://www.syndex.org/*

[9] Acceleo, *http://www.eclipse.org/acceleo/*

[10] R.Bucher, L.Dozio, CACSD under RTAI Linux with RTAI-LAB, Real Time Linux Workshop, 2003.

[11] P.Gai, E.Bini, G.Lipari, M.Di Natale, L.Abeni Architecture for a portable open source real-time kernel environment, Real-Time Linux Workshop, 2000.

[12] Erika Enterprise, *http://www.erika.tuxfamily.org*

[13] R.Bucher, S.Mannori, P.Gai, Scicos-FLEX Code Generator, *http://erika.tuxfamily.org/drupal/scilabscicos.html*

[14] QRTAILab, *http://qrtailab.sourceforge.net/*

[15] A.Guiggiani, M.Basso, M.Vassalli, F.Difato, Realtime Suite: a step-by-step introduction to the world of real-time signal acquisition and conditioning. Real-Time Linux Workshop, 2011. *http://www.rtaixml.net/*

[16] M Di Natale, L Guo, H Zeng, A Sangiovanni-Vincentelli Synthesis of multitask implementations of simulink models with minimum delays, Industrial Informatics, IEEE Transactions on 6 (4), 637-651

[17] AdaCore: 'Project P' and 'Hi-MoCo' Research Projects Launched, web site http://www.adacore.com/press/project-p-and-hi-moco/

[18] D. Harel. "Statecharts: A Visual Formalism for Complex Systems," in *Science of Computer Programming*, 8(3):231–274, June 1987.

[19] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Guernic, and R. de Simone. "The synchronous languages 12 years later," in *Proceedings of the IEEE*, 91, January 2003.

[20] C. Andre, SyncCharts: A Visual Representation of Reactive Behaviors, Technical Report RR 96-56, I3S, Sophia Antipolis, Avril 1996.

[21] *The Mathworks Simulink and StateFlow User's Manuals*, Mathworks, web page: http://www.mathworks.com.

[22] M. Otter, K.-E. Arzen, I. Dressler StateGraph-A Modelica Library for Hierarchical State Machines, 4th International Modelica Conference, Hamburg, March 7-8, 2005

[23] UML 2.0 Specification, available at http://www.omg.org