Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science
Espoo 2004 TKO-C93/04

Proceedings of Seminar on Real-Time Programming

Vesa Hirvisalo (ed.)

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science
P.O. Box 5400
FIN-02015 HUT
Espoo
Finland

Keywords: real-time programming, programming tools

ISBN: 951-22-7456-6
ISSN: 1239-6907

Preface

Real-time programming is an art of creating computer software, whose timing behavior is predictable. Typically, real-time software systems are embedded systems, and vice versa. Such systems are very common – it has been estimated that 99% of all software applications are embedded. Real-time systems include communication systems, signal processing systems, control systems, databases, etc. They vary from small intelligent cards (or tags) though automobiles and airplanes to huge system, e.g., those used by military forces in battlefields.

Real-time software systems are often very complex. Typically, they consist of a number of interacting processes that must be synchronized in a predictable way. It is hard to implement the processes in a way that makes task execution times predictable. Because of such problems, real-time programming seems inherently difficult. Productivity of programming can be very low. For safety-critical applications (for which timing guarantees are a must), the productivity of the programming work can be a few lines of code per a day and a programmer.

Programming language development was very active during the 1970's and 1980's. Real-time programming was also considered. A number of programming languages were developed. The languages concentrated on the various aspects of real-time programming, e.g., worst-case execution time guarantees or incorporating process and synchronization abstractions.

Most of the languages developed for real-time programming did not reach application programming or have died away. Only Ada seems to have survived – and, its future is somewhat unclear, too. The current way of writing real-time applications seems to be the use of C language with process and synchronization mechanisms of a real-time operating system (RTOS). The number of real-time operating systems used is huge, and many of them do not conform with any standard (e.g., RT POSIX).

The Laboratory of Information Processing Science at Helsinki University of Technology organized a seminar on real-time programming during autumn 2004. The goal of the seminar was to discuss the options available to real-time programming (in addition to the C&RTOS combination). These proceedings include some of the work presented in the seminar.

The seminar consisted of three sessions. The first session of the seminar discussed Real-Time Java, which seems to be the current hot topic inside real-time programming. The second session discussed Ada and object-oriented programming, because the new Ada 2005 work is becoming available. For the third session, we wanted to have a specialized aspect of programming (to be compared with the general purpose programming languages). Because of the success of statecharts and the availability of tools generating production-quality code from statecharts, state-based programming was selected as the topic of the third session.

Espoo, December 2004
Seminar chair

Vesa Hirvisalo

Contents

Introduction to Real-time Java

Aleksi Ahtiainen
Helsinki University of Technology
Laboratory of Information Processing Science
P.O.Box 5400, FI-02015 TKK, Finland
Aleksi.Ahtiainen@tkk.fi

Abstract

Since the late 1990s several real-time extensions to Java programming language have evolved. RTSJ, the Real-time Specification for Java, has been the main subject of research and implementation work. This article introduces the current contents, future challenges and existing implementations of the RTSJ. Other real-time related Java technologies and their relationship with RTSJ are then introduced briefly.

1 Introduction

Java programming language has achieved its major success as a general purpose portable programming language. It is not applicable for real-time programming as such. Reason for this is Java's lack of real-time features like accurate timing, asynchronous transfer of control and predictability. The lack is currently actively being solved by specififfying extensions to Java.

Purpose of this article is to introduce the current research and specification work of real-time programming in Java. The article is aimed at computer scientists and engineers who are planning to use Java for real-time applications or are planning to join the real-time Java research community. The article assumes that the reader already has basic understanding of real-time systems and experience in Java programming.

Second chapter introduces and demonstrates one real-time Java related technology, real-time specification for Java (RTSJ), in detail. The third chapter provides an overview of the future challenges and possible research areas related to the RTSJ. Currently existing implementations of the RTSJ are described in chapter four. Chapter five introduces other real-time Java related technologies and their relation to the RTSJ. The article is concluded in chapter six.

2 The Real-time Specification for Java (RTSJ)

The real-time specification for Java defines a class library and a virtual machine extension to Java. One of the main goals of the RTSJ specification is to add real-time capabilities to Java without any additions to the language itself. The aim is to create a specification with good backwards compatibility and minimal semantical changes. The specification also defines interaction features between non-, soft- and hard real-time threads within the same virtual machine. This is important since it is very common to have real-time constraints only for parts of an application. [15]

RTSJ is developed as Java community process (JCP) and version 1.0 [10] was published in 2001 as a Java specification request (JSR). Version 1.0.1 [11], which mainly clarifies some semantics, was published and accepted by the JCP working group in August 2004. Its final release is still pending on the reference implementation (RI) and technology compatibility kit (TCK) updates.

Figure 1 describes the main new Java classes introduced by RTSJ. Here is a short summary of the main RTSJ features. All of them are described in more detail in the following subsections. [10]

- Memory management with scoped memory regions
- Accurate time and execution clock
- Schedulable objects, a priority scheduler for them and a framework for dynamic feasibility analysis in schedulers
- Asynchronous interrupts and asynchronous transfer of control
- Priority inheritance algorithm for preventing priority inversion of synchronized blocks
- Direct memory access

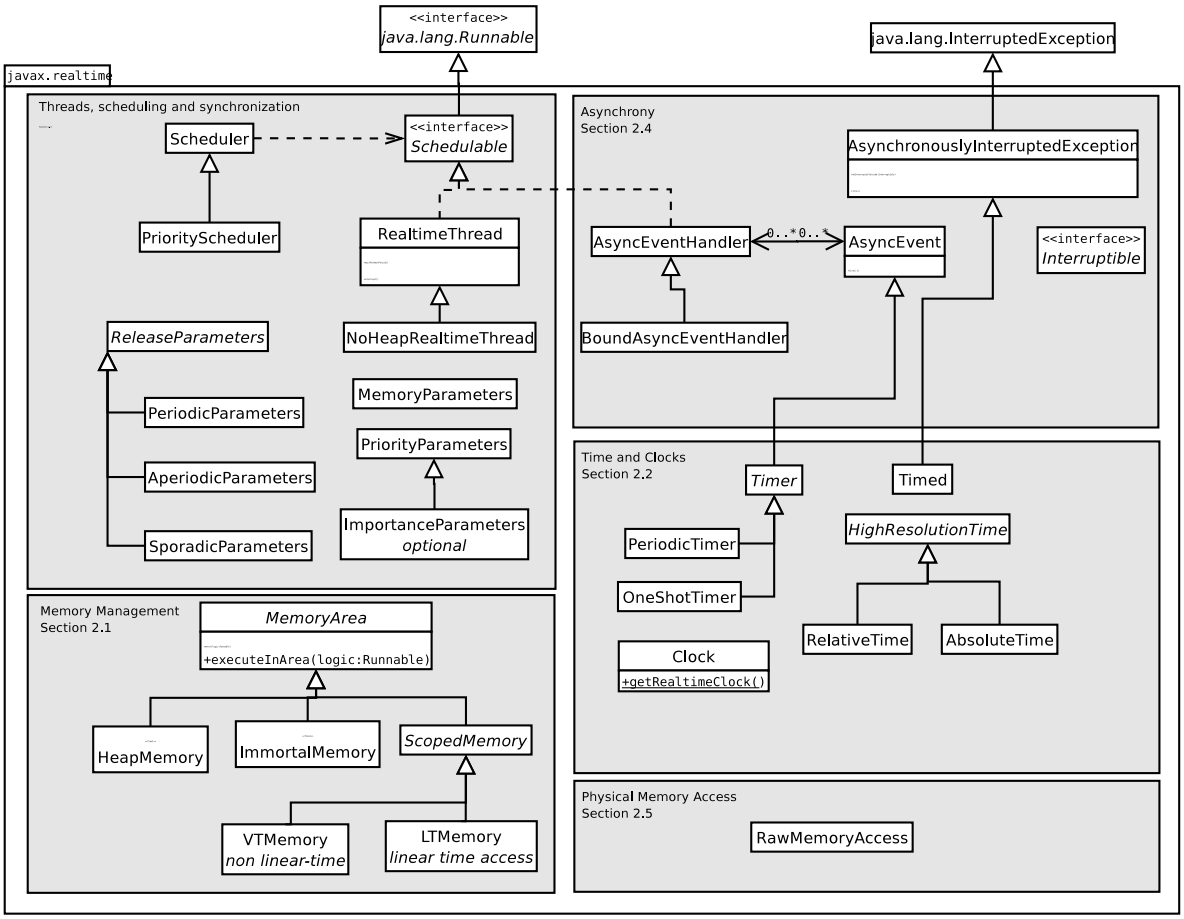


Figure 1. Main RTSJ classes

2.1 Memory Management

Perhaps the most controversial new feature of RTSJ is the introduction of scoped memory regions. Scoped memory regions are separate memory areas outside the garbage collected heap memory. Therefore real-time dynamic memory allocation can be done without worrying about garbage collection delays. The scoped memory regions are similar to method call stacks, but they are handled explicitely. The idea is that threads can enter and exit scopes and when all threads have exited a certain scope all the allocated objects in the scope are freed by the virtual machine. The scopes can also be nested within other scopes.

All the memory region types are represented by abstract `MemoryArea` class and the scoped regions are represented by its abstract subclass `ScopedMemory`. The two actual subclasses of `ScopedMemory` are `LTMemory` for memory allocations which assumes linear time complexity of allocations and `VTMemory` for the situations in which linear time is not a necessity. Code can be run in a specified scope by incorporating the code in a new `Runnable` object and passing it as a parameter to `MemoryArea.enter()` or `MemoryArea.executeInArea()` methods.

It is still possible to use the garbage collected heap memory in non real-time parts of RTSJ applications. The heap memory is represented by a singleton subclass of `MemoryArea` called `HeapMemory`. There is also a memory region represented by subclass `ImmortalMemory` which can be used for static allocation of objects that are never freed while the application is running.

The semantics of the scopes are somewhat complicated, because referential integrity must be assured by the JVM. Therefore it is only possible to have memory references from the nested scopes to the parent scopes and not vice versa. References from immortal or heap memory to scoped memory regions are also forbidden.

2.2 Time and Clocks

Common Java time classes are not accurate enough for real-time applications. Therefore RTSJ introduces new classes with accuracy up to nanoseconds. There are separate classes for relative time (`RelativeTime`) and absolute time (`AbsoluteTime`). The new time classes can be used in timing events and also with `sleep()` in the `RealtimeThread` class. The virtual machine's global real time clock is always available through call to `Clock.getRealtimeClock()`.

There are a couple of ways to time events to happen at specified time in the future. The new `Timed` class is a subclass of `AsynchronouslyInterruptedException` and with it timers can be fired in the RTSJ's asynchronous transfer of control (ATC) framework. RTSJ's more advanced event handling framework can be used with class `Timer`'s two subclasses `OneShotTimer` and `PeriodicTimer`. They all inherit from the `AsyncEvent` abstract class. Asynchronous transfer of control and event handling are described in more detail in section 2.4.

2.3 Threads, Scheduling and Synchronization

RTSJ introduces the concept of schedulable objects which implement the new `Schedulable` interface. Schedulable objects can be threads or event handlers. The threads are implemented in two different classes, one for soft real-time threads (`RealtimeThread`) and another for hard real-time threads (`NoHeapRealtimeThread`). The latter is not allowed to use the garbage collected heap memory and therefore its execution deadlines are never missed due to garbage collection delays. The scheduler is also always able to pre-empty possible background garbage collector when `NoHeapRealtimeThread` needs to be run, because it does not use the garbage collector. Also RTSJ's event handler class, `AsyncEventHandler` implements the `Schedulable` interface (see section 2.4).

Three types of parameters are passed to new schedulable objects at construction. The parameters are specified using separate placeholder classes. Class `MemoryParameters` specifies memory usage limits for the new thread and subclasses of `SchedulingParameters` specify the thread's priority. Subclasses of `ReleaseParameters` can be used to specify the timing limits. The timing limits can define the schedulable object either as periodic, aperiodic or sporadic (classes `PeriodicParameters`, `AperiodicParameters` and `SporadicParameters` respectively).

The last method call in periodic threads must be `waitForNextPeriod()`, which blocks the thread until the start of its next execution. Depending on the scheduler implementation, the method can return an indicator that a deadline miss has occurred, and the application can then react accordingly. Unfortunately there is no such way to detect deadline misses of aperiodic or sporadic threads.

RTSJ defines a framework for schedulers with dynamic feasibility analysis but does not specify an implementation. RTSJ specifies one scheduler which every implementation must implement. This scheduler is a priority scheduler with priority inheritance algorithm for priority inversion prevention. A scheduler with priority ceiling algorithm is defined in the specification, but it is optional to implement.

Objects in hard real-time and other domains of an application often need to communicate with each other. Straightforward method call communication would introduce possible garbage collection delays into the hard real-time domain and this must not be allowed. Therefore RTSJ specifies non-blocking queues for this communication (`WaitFreeDequeue`, `WaitFreeReadQueue` and `WaitFreeWriteQueue`).

2.4 Asynchrony

RTSJ defines two basic asynchrony methods, asynchronous event handling and asynchronous transfer of control (ATC). The event handling framework is implemented by asynchronous events of class `AsyncEvent`. They can be bound to several schedulable objects, `AsyncEventHandler`s. All the handlers are called when the `AsyncEvent`'s `fire()` method is called. The virtual machine may also implement event handling for external events like physical interrupts. These can be bound to `AsyncEvent`s by using its method `bindTo()`. By default the virtual machine is allowed to use a single thread for several `AsyncEventHandler`s. Class `BoundAsyncEventHandler` can be used, if the application needs a dedicated event handler thread for some event type.

ATC is needed in order to have fast response times to certain events, like error situations or application mode changes. Also some decision making can be time-dependent. For example the application might be allowed to iteratively calculate better results until an external event happens. RTSJ implements ATC within the Java exception handling framework. Common Java threads do not guarantee immediate stop of thread execution when the thread's `interrupt()` method is called. RTSJ's `RealTimeThread.interrupt()` method does guarantee immediate handling of exception `AsynchronouslyInterruptedException` when it is called from some other thread. The methods which support this behavior must be identified by "`throws AsynchronouslyInterruptedException`". If the `interrupt()` method is called when the thread to be in-

terrupted is outside such method, the interruption is handled immediately after the thread enters or returns to another method with the `throws` clause.

RTSJ also specifies a somewhat more structural way of specifying threads as asynchronously interruptible. A class can implement the `Interruptible` interface and define method `interruptAction()` accordingly. After this the interruptible thread can be started by calling `doInterruptible(Interruptible intThread)` method of some `AsynhcronouslyInterruptedException` object. Then the `interruptAction()` method is called by the virtual machine immediately whenever the asynchronously interrupted exception's `fire()` method is called.

2.5 Physical Memory Access

Many real-time applications are embedded applications that must have access to physical memory in order to implement hardware interfaces like drivers. RTSJ specifies a restricted way of accessing physical memory regions supported by the virtual machine implementation. In addition to regular MMU memory the virtual machine can specify regions for special purposes, like IO vectors. Is is only possible to use primary types (byte, int, etc.) in the raw physical memory. Class `RawMemoryAccess` specifies the methods for accessing all the basic types.

2.6 Example Program

There is a simple example RTSJ program in figure 2 on page 8. The program illustrates some of the RTSJ features. It creates a new periodic thread and illustrates how the method `RealTimeThread.waitForNextPeriod()` can be used to wait for the next period and to notice a possible deadline miss.

3 Future Work and Challenges of RTSJ

RTSJ is not a complete or optimal solution to all Java real-time programming needs. Some completely different approaches have been specified in other real-time Java technologies (see section 5) and several future challenges have been recognized in the current RTSJ [15] as described below. In addition to specification work, efficient implementations of RTSJ are naturally needed, if it is to gain widespread utilization. Compilers with static analysis for scoped memory regions and run-time environments with minimized ATC- and event handling overheads are a definite necessity.

The scheduling framework in RTSJ lacks many features, which would be beneficial for real-time development. Many applications use several different schedulers at the same time. RTSJ does not currently take this into account. It could incorporate some communication methods between simultaneous schedulers. The specification also does not handle deadline misses of aperiodic or sporadic threads, even though they might often occur especially in soft real-time applications. In many situations it would also be helpful, if some separate thread importance parameter could be used when the scheduler needs to decide, which deadline is the least harmful to miss. The dynamic feasibility analysis framework in RTSJ would also gain strength from having additional clocks, like CPU time and thread blocking time, specified. [15]

RTSJ does not currently specify any support for distributed or multiprocessor systems, even though these are important real-time application areas. There has been some work on specifying a real-time Java RMI framework [12], but no implementation currently exists.

Probably the biggest challenges in RTSJ are related to the introduction of scoped memory regions. It is difficult to use existing Java libraries with scoped memory. Referencing scoped memory has been recognized also as one of the biggest challenges in real-time RMI specification work. [12].

Some new scoped memory design patterns have been designed ([18] and [16]) in order to make it easier for Java programmers to start working with scoped memory. The main difficulty is the restriction of not having references from immortal or higher scopes to lower ones. It might be good to have some class libraries for handling this kind of weak references in RTSJ. [15] Programming could also be made easier by having separate reference counting scoped memory areas or recycling list classes specified in the RTSJ. [15]

In the future the need for scoped memory might be completely removed by efficient hard real-time garbage collection. This has already been implemented in the JamaicaVM RTSJ implementation. Better static compile time analysis would also make it possible to reserve space for dynamic allocation of objects directly in the method call stack. Then the scoped memory regions would be completely unnecessary. [17]

4 RTSJ Implementations

Several RTSJ implementations already exist and are also under constant development as the specification and developer needs evolve. The Java Community Process, which RTSJ specification has been using, demands a reference implementation (RI) and also a technology compatibility kit (TCK), which can be used to test other implementations. TimeSys has the task to implement these for RTSJ [6]. Their RI and TCK have been developed to run compliantly on

TimeSys' own Linux distribution and also run with some hard real-time functionality loss in other Linux distributions.

TimeSys has also a commercial Linux implementation of RTSJ, called JTime. [7] In addition to the mandatory RTSJ features, it includes support for the dynamic feasibility analysis as specified in the RTSJ scheduling framework. The implementation is currently only supported with TimeSys Linux on PowerPC and IA-32 (x86) processor architectures.

JamaicaVM is another commercial RTSJ implementation. [19] In addition to RTSJ requirements it has hard real-time garbage collector. [17] Unlike TimeSys' implementation, JamaicaVM has been ported to several specialized real-time operating systems (VxWorks, Linux/RT, embOS, QNX, EUROS, ThreadX) and several processor architectures (x86, PowerPC, Sparc, RC32, StrongARM, NEC v850, Net+ARM). It also runs as non real-time virtual machine on common Linux, SunOS and Windows, which makes development and functionality testing easier.

GNU GPL licensed RTSJ implementation, jRate, is also under development. [14]. It is implemented as a compiler front-end and extended runtime system to the Gnu Compiler for Java, GCJ [2]. It is a research project which is being developed on x86 Linux, but should run on any POSIX compliant operating system.

Naturally Sun Microsystems also has a big role in the possible widespread adoption of RTSJ. Sun currently does not have its own RTSJ implementation, but is developing RTSJ JVM on top of a HotSpot JVM in project Mackinac [8]. Sun Labs, NASA's Jet Propulsion Labs and Carnegie Mellon University are also cooperating in an RTSJ adoption project Goldengate [3]. The project aims to implement a software called Mission Data System for use in future Mars missions.

5 Other Java Real-time Technologies

In addition to RTSJ several other real-time Java technologies have evolved since the late 1990s. There is no established standard yet. The following sections shortly introduce some of these technologies and their relationship with the RTSJ.

5.1 Java 2 Micro Edition (J2ME)

Java 2 Micro Edition does not specify real-time features, but due to its current importance in embedded Java development, it deserves a short introduction here. Idea of Sun's J2ME specification work is to standardize a subset of parts of the Java language and libraries which are usable in embedded devices like mobile phones. Main goal has been to minimize the memory consumption of the Java virtual machine. [5]

J2ME has been a successful technology and many companies have added their own real-time capabilities into their J2ME implementations. Sun's own CDC HotSpot (former C virtual machine, CVM) J2ME implementation does not provide real-time capabilities, though [4].

5.2 Ravenscar Profile for Java

Ravenscar profile for Java is a highly restricted subset of RTSJ for high-integrity applications, like nuclear reactors. Its goal is to improve predictability, integrity and security of real-time Java by removing features with high overhead and complex semantics. The specification does not go as far as removing multi-threading features, since they are analytically well understood nowadays. Somewhat obscure name of the specification is inherited from similar ADA profile. Like RTSJ, Ravenscar profile tries to keep programs runnable also in pure RTSJ virtual machines. [20]

The profile divides program execution to separate initialization and mission phases. The former initializes the threads after which they are started and no more initialization is allowed. The memory management is much more restricted, garbage collection is not included and only immortal and linear-time memory allocation is allowed in mission phase. Also the complexities of scoped memory have been removed by removing scope nesting and scope sharing between threads.

Only periodic and sporadic threads (not asymmetric) are allowed and the framework for dynamic feasibility analysis has been dropped. Static feasibility analysis must be performed prior to running the application, and therefore there is no need for overrun or deadline miss handling. The fixed priority scheduler must implement priority ceiling emulation in order to prevent priority inversion when using synchronized blocks of code. Java's `wait()`, `notify()`, `notifyAll()` and `sleep()` methods have been completely removed. Also asynchronous transfer of control and timers have been removed from RTSJ in order to simplify scheduling.

The profile also defines several clarifying restrictions on the language constructs, for example expression evaluation must not depend on evaluation order. Therefore a specialized compiler with plenty of static code analysis is needed to implement the profile. Naturally also a high integrity Ravenscar Java virtual machine needs to be used.

5.3 J Consortium's Real-Time Core Extensions for Java

J Consortium published a competing real-time extensions to RTSJ in 2000. [22] There are no implementations of these core extensions yet. The main features in comparison to RTSJ are: [21]

- Focus on hard real-time instead of combining the soft and hard real-time domains like RTSJ does
- Attach to existing virtual machines instead of changing Java semantics
- Direct sharing of objects between hard real-time and traditional Java domains is specified according to design of ADA's protected objects
- Does not support the immortal or scoped memory areas like RTSJ. Instead specifies allocation regions with explicit allocations and deallocations. Also stack allocation can be used with static analysis support at the compile time.
- Performance oriented, the goal of the specification work has been to offer speed comparable to C and C++ in addition to predictability.

Ravenscar-like high integrity profile is also being designed for core extensions, but it has not yet been published. [20]

5.4 Open Group's Real-time Java Standardization Work

Due to the competing specifications, Open Group has been given the task to create a new safety-critical Java specification. The goal is to unify between the RTSJ and the Core extensions. Target of the work is to submit a Java specification request (JSR), reference implementation (RI) and technology compatibility kit (TCK) to the Java community process (JCP) by second quarter of 2005. [13] There are no public reports on the progress yet.

5.5 Stand-alone Real-time Java Products

Prior to the current specification work some companies have published their own real-time Java solutions. For example PERC [9] embedded virtual machine provides some real-time capabilities but does not conform to the RTSJ. Ajile Java processor [1] provides on-chip Java virtual machine with support for only a subset of the RTSJ specification.

6 Conclusions

This paper has reviewed the current work on extending Java for real-time programming. It has described the real-time specification for Java (RTSJ) and shortly introduced other real-time Java technologies.

Java is not perfectly suitable programming language for real-time applications, but there is enough interest in developing it into one. Currently the real-time specification for Java (RTSJ) has the most momentum and has already several implementations. Existence of other specifications and implementations illustrates that the RTSJ is not the only solution to the difficulties introduced by real-time demands. Some new features of the RTSJ, especially scoped memory regions, have been shown to add plenty of semantical complexity in programming. Therefore Java programmers need to design and adopt new programming practices, at least until real-time garbage collectors get efficient enough to be used instead.

Acknowledgements

The author would like to thank T-106.850 Seminar on Real-time Programming tutor Vesa Hirvisalo and course students for their helpful comments on the drafts of this paper.

References

[1] aJ-100 Java processor. At `http://www.ajile.com/`, Referenced 2004-OCT-11.

[2] GCJ - GNU compiler for Java. At `http://gcc.gnu.org/java`, Referenced 2004-OCT-11.

[3] Goldengate project. At `http://research.sun.com/projects/goldengate/`, Referenced 2004-OCT-11.

[4] J2ME CDC hotspot implementation. At `http://java.sun.com/products/cdc-hi/index.jsp`, Referenced 2004-OCT-11.

[5] Java 2 platform, micro edition (J2ME). Published at `http://java.sun.com/j2me/`, Referenced 2004-OCT-11.

[6] Java reference implementation (RI) and technology compatibility kit (TCK). At `http://www.timesys.com/index.cfm?bdy=java_bdy_ri.cfm`, Referenced 2004-OCT-11.

[7] JTime. At `http://www.timesys.com/index.cfm?bdy=java_bdy.cfm`, Referenced 2004-OCT-11.

[8] Mackinac project whitepaper. At `http://research.sun.com/projects/mackinac/mackinac_whitepaper.pdf`, Referenced 2004-OCT-11.

[9] PERC. At `http://www.aonix.com/perc.html`, Referenced 2004-OCT-11.

[10] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, and R. Belliardi. *The Real-Time Specification for Java*. Addison-Wesley, 2001. Published at `http://www.rtj.org/`.

[11] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, R. Belliardi, A. Wellings, and D. Holmes. The real-time specification for Java, 2004. Published at `http://www.rtsj.org/specjavadoc/book_index.html`, Referenced 2004-OCT-11. This version of the specification has been approved as a minor change on Aug 9 2004 but will not be final until Reference Implementation and Technology Compatibility Kit have been updated.

[12] A. Borg and A. Wellings. A real-time RMI framework for the RTSJ. In *Proceedings of the 15th Euromicro Conference on Real Time Systems*. Euromicro, IEEE, Jul 2003.

[13] J. Child. Real-time Java takes aim at embedded control. *RTC Magazine*, (August 2004), 2004. Published at `http://www.rtcmagazine.com/home/article.php?id=100111`, Referenced 2004-OCT-11.

[14] A. Corsaro. jRate. At `http://www.cs.wustl.edu/~corsaro/jRate/`, Referenced 2004-OCT-11.

[15] P. Dibble and A. Wellings. The real-time specification for Java: current status and future work. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pages 71–77. IEEE Computer Society, 2004.

[16] P. F., F. J. M., H. D., and J. Vitek. Real-time Java scoped memory: Design patterns and semantics. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pages 101–110. IEEE Computer Society, 2004.

[17] S. F. The impact of realtime garbage collection on realtime Java programming. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pages 33–40. IEEE Computer Society, 2004.

[18] B. G., C. T., C. V., C. V., D. D., G. B., I. M, M. K., M. A., and R. K. Programming with non-heap memory in the real time specification for Java. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 361–369. ACM Press, 2003. Published at `http://doi.acm.org/10.1145/949344.949443`.

[19] a. GmbH. JamaicaVM realtime Java technology. At `http://www.aicas.com/`, Referenced 2004-OCT-11.

[20] J. Kwon, A. Wellings, and S. King. Ravenscar-Java: a high integrity profile for real-time Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140. ACM Press, 2002. Published at `http://doi.acm.org/10.1145/583810.583825`.

[21] K. Nilsen and A. Klein. *OTM Workshops 2003, Lecture Notes in Computer Science*, volume 2889, chapter Issues in the Design and Implementation of Efficient Interfaces between Hard and Soft Real-Time Java Components, pages 451–465. Springer-Verlag Berlin Heiderlberg, 2003.

[22] J. C. Real-Time Java Working Group. Real-time core extensions, 2000. Published at `http://www.j-consortium.org/rtjwg/index.shtml`, Referenced 2004-OCT-11.

```
import javax.realtime.*;

public class PeriodicThread extends RealtimeThread
{
  public boolean done;

  public PeriodicThread(PriorityParameters pri,
                        PeriodicParameters per)
  {
    super(pri, per);
    this.done = false;
  }

  public void run()
  {
    int c = 0;
    boolean deadlineNotMissed = true;

    while (!done && deadlineNotMissed) {
      System.out.println("Counter at "+c);
      // Do other periodic actions
      deadlineNotMissed = this.waitForNextPeriod();
    }

    if (!deadlineNotMissed) {
      // Period deadline has been missed, handle the problem
    }

    System.out.println("Periodic thread is exiting");
  }

  public void setDone()
  {
    this.done = true;
  }
}

class SimpleExample
{
  public static void main(String[] args) throws InterruptedException
  {
    // Create and start a new periodic thread with 100 ms period
    PeriodicThread t =
      new PeriodicThread(new PriorityParameters(PriorityScheduler.MAX_PRIORITY),
                         new PeriodicParameters(null, new RelativeTime(100, 0),
                                                null, null, null, null));
    t.start();
    Thread.sleep(50000);
    t.setDone();
  }
}
```

Figure 2. An Example RTSJ Program

Threads and Scheduling in Real-Time Specification for Java Framework

Ville Likitalo
Helsinki University of Technology
Laboratory of Information Processing Science
liki@iki.fi

Abstract

This paper describes Real-Time Specification for Java[1] *compliant virtual machine in general. Emphasis is around threads and scheduling with other areas only briefly mentioned. Real-Time requires some fundamental differences compared to normal Java, also most RT systems are embedded as well.*

1. Introduction

Real-Time Specification for Java extends Java ™virtual machine, making it possible to build Real-Time systems in Java. RTSJ specifies strict requirements on schematics of Real-Time operation but still maintains backward compatibility so that normal Java programs will work with RTSJ-compliant virtual machine aswell.

RTSJ specifies framework allowing scheduling required by real-time operation. Many details are left to be specified by implementation but the specification is strict enough that real-time requirements can be met but at the same time the specification is open enough to allow tailoring for specific problems or architectures.

2. Real-Time Specification for Java

Real-Time Specification for Java defines extensions necessary to facilitate Real-Time systems. RTSJ does not inhibit backward compatibility, normal Java-programs will also execute under RTSJ virtual machine. Also portability - One of the key features of Java - is to be preserved, but not at the expense of predictability of execution. [8]

2.1. Design goals

Primary design goals of RTSJ are the following: [8]

- Predictable execution
- extensibility
- No semantic extensions to the Java language
- To allow variations in implementations

Secondary goal is “Write Once, Run Anywhere”, in other words binary portability. [8]

2.2. Scheduling

Because of different requirements in RT systems for scheduling and dispatching models RTSJ does not dictate scheduling. One base scheduler is required by the specification, but the specification is flexible enough to allow development of different models. [8]

2.3. Memory management

RTSJ regards automatic memory management as an important feature of Java. Definition of Garbage Collector [2] is open to accommodate any GC algorithm, but effects on execution, preemption and dispatching must be taken into account. [8]

3. Virtual Machine

First virtual machine[3] was a time-sharing system that offered multiprogramming and an extended machine with a more convenient interface than the bare hardware. Nowadays a modern Java VM is a bit different, but RTSJ VM resembles the original counterpart more. [15]

Normal Java VM running as a single process in host operating system[4] is at the mercy of the host OS while trying

[1] RTSJ

[2] GC
[3] VM
[4] OS

to guarantee execution times. Many RT systems are embedded and VM runs as sole process on top of minimalistic OS, or it is the OS itself. [1, 3]

VM offers the same benefits for RT systems with the addition of predicability of execution. But naturally correctness of the VM itself is hard to prove. [1, 8]

4. Time

Time is fundamental measure in RT systems and the normal accuracy of milliseconds in Java is inadequate for RT systems. RTSJ defines presentation of Time in accuracy of nanoseconds, and also makes distinction between absolute and relative time. All time objects must maintain accuracy of nanoseconds and provide simple arithmetic operations. [8]

Further RTSJ provides RationalTime for efficient presentation of occurances during period, i.e. frequency. [8]

5. Threads

Traditionally each process has an address space and a single thread of control. A thread has its own program counter and stack, but shares other resources with other threads within a process. [15]

RTSJ defines 2 types of threads that extend java.lang.Thread but have more precise scheduling semantics. Due to nature of RT systems, threads must be able to allocate memory from other areas than heap and provide semantics for handling asynchronous events. Extensions provided by RTSJ must only be available in the context of RealtimeThread, NoHeapRealtimeThread, AsyncEventHandler or BoundAsyncEventHandler. [8]

Main distinction between RealtimeThread and NoHeapRealtimeThread is that the later may not reference memory or objects in memory that is dynamically allocated, ie. maintained by GC. This allows NoHeapRealtimeThread to have higher priority than GC and interrupt execution of GC. [8]

The separation could also have been made by preserving priorities higher than GC for NoHeapRealtimeThreads and lower than GC for RealtimeThreads, but this does not place any restrictions on the priorities and illegal references can be detected at the time of compilation.

5.1. RealtimeThread

RealtimeThread extends java.lang.Thread to hold parameters necessary for RT operation. Priority-preemptive dispatching model in Java is similar enough to the model used in many RT operating systems, but semantics were relaxed to allow execution on a wide variety of operating systems. [8]

5.2. NoHeapRealtimeThread

NoHeapRealtimeThread is a specialized form of RealtimeThread. It is provided for time-critical applications that must have priority higher than GC. Therefore logic contained within NoHeapRealtimeThread may never allocate objects from heap, or reference objects allocated in the heap. This way it has been made safe for NoHeapRealtimeThread to interrupt execution of GC. [8]

6. Synchronization

Avoiding priority inversion when sharing resources is very important in RT systems. Therefore RTSJ requires implementations to strengthen the schematics of the Java synchronized keyword. At least one algorithm that prevents priority inversion is required. RTSJ specifies the following algorithms. [8]

- Priority Inheritance
- Priority Ceiling Emulation

RTSJ defines that threads waiting on a mutex must be priority ordered, with threads with same priority behaving in FIFO[5]-manner. [8]

Also, RTSJ specifies wait-free queues for communication from RTSJ-scope with the traditional Java parts of a system. [8]

6.1. Priority Inversion

Priority inversion in this case refers to a situation where a thread with higher priority is waiting on a mutex reserved by a thread with lower priority, but a thread in ready state which has higher priority than the thread reserving the mutex prevents the thread from running and releasing the mutex. [15]

6.2. Priority Inheritance

Priority Inheritance works by modifying priority of a thread owning the mutex. If a thread with a higher priority is waiting for the mutex, priority of the thread owning the mutex is raised to priority of the waiting thread if it was lower. After the thread releases the mutex its priority is restored back to normal. This ensures that a thread with lower priority may no longer prevent the thread owning the mutex from running. [11]

[5] First In, First Out

6.3. Priority Ceiling Emulation

In Priority Ceiling Emulation a mutex is assigned a highest possible priority it is valid in. When a thread that has a priority lower than this set limit locks the mutex its priority is raised to the limit set for the mutex. When the mutex is released the priority of the thread is restored back to normal.

7. Scheduling

Scheduling means that when two different processes, or threads are at ready state, a scheduler must choose which one to run next. Events in RT system can be categorized as periodic or aperiodic. Scheduler must take care that time restrictions set upon the system are met. A RT system that meets the criteria in 1 is said to be schedulable. [15]

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1 \quad (1)$$

Where event i occurs with period P_i and requires C_i seconds of CPU time to handle each event.

There are two basic types of schedulers, static and dynamic. Static schedulers make scheduling decisions beforehand. This means that information about occurring events and work needed must be exact so that the scheduler can be tailored for the system. Dynamic schedulers do not suffer from this restriction but are more complex. [15]

7.1. Scheduling in RTSJ

Scheduler required by RTSJ is fixed-priority preemptive, however other schedulers may be provided. Schedulable objects are presented by RealtimeThread, NoHeapRealtimeThread and AsyncEventHandler or any object implementing Schedulable interface. RTSJ requires additional lower priorities that will be assigned to native Java threads. By default new threads inherit SchedulingParameters from their parent thread. [8]

It is also even possible to introduce several schedulers into one system, with perhaps different schedulers with each scheduling one type of tasks. However RTSJ does not specify any methods with which these different schedulers could communicate with each other. [8]

7.2. PriorityScheduler

PriorityScheduler is the base scheduler required by RTSJ. Base scheduler is fixed-priority preemptive scheduler with 38 unique priority levels, 28 required for RT threads and rest for native Java threads. The following requirements are set: [8]

- Preempted threads may be placed in the appropriate queue at any position.
- A blocked thread that becomes read is added to the tail of any runnable queue for that priority.
- A thread whose priority is changed is added to the tail of the runnable queue for the new priority level.

The very exact manner in which queues are operated is left to the implementation, but RTSJ requires that the exact operation is documented. [8]

7.3. SchedulingParameters

Subclasses of SchedulingParameters provide parameters used by scheduler. RTSJ specifies PriorityParameters and ImportanceParameters and other classes may be specified if required by particular implementation of scheduler. [8]

Instance of PriorityParameters can be assigned for threads managed by schedulers that use a single integer[6] to determine execution order. [8]

Instance of ImportanceParameters can be assigned for threads to determine additional scheduling metric used in addition to priority level. This metric can be used e.g. in overload conditions to determine threads that are more important than others. RTSJ does not require base scheduler to take importance into account, but it is recommended that a simple extension is provided. [8]

7.4. ReleaseParameters

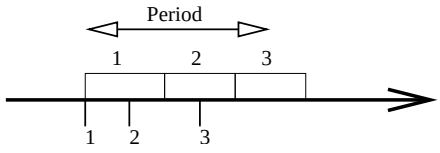
ReleaseParameters is an abstract class defining release characteristics of threads. ReleaseParameters holds execution cost, deadline and overrun and miss handlers. Overrun handler is invoked if invocation of an object exceeds its assigned cost and miss handler if an object is still executing after its deadline has passed. RTSJ specifies PeriodicParameters, AperiodicParameters and SporadicParameters. [8]

7.5. Periodic events

PeriodicParameters indicates that a call to waitForNextPeriod() will be unblocked at the start of a each period, thus the object gets periodically executed. PeriodicParameters takes start time and length of the period as additional parameters. [8]

To schedule periodic events a timer must be set to match the period. Thread handling the periodic event must call waitForNextPeriod() and becomes blocked until timer unblocks the call after the period has exceeded. Overrun handler may be called if execution cost exceeded cost set in the

[6] Priority level



parameters and deadline miss handler must be called if the deadline in set in the parameters was exceeded. [8]

7.6. Aperiodic events

AperiodicParameters characterizes an object that may become active at any time. Aperiodic events are usually triggered via interrupt by external events or internally in the program. [8]

For aperiodic threads the call into fire() is made when the launching event occurs. Again, overruns may be detected but deadline misses must be detected. [8]

7.7. Sporadic events

SporadicParameters is a notice to the scheduler that the object will release execution aperiodically but with a minimum time between releases. [8, 5]

Sporadic threads are handled in the same way as aperiodic threads, but due to nature of the event implementations are required to maintain to list of times when the event has occurred. For sporadic threads detection of deadline misses is not required. [8]

8. Feasibility Analysis

For static schedulers no new schedulable events are accepted during operations and feasibility analysis can be performed beforehand. That may include complex statistical analysis and simulation of the system, however dynamic schedulers are required to perform the analysis in real-time and no such methods are feasible nor data present. [8, 15, 5]

However, noting that applications are required to provide execution cost for real-time events and recalling 1 we note that the feasibility analysis is trivial for periodic events as the period must be known. Aperiodic and sporadic events can be added into the analysis by some elementary probability calculus provided that their probability is known. Although this means that the analysis is not proof for any selected period like illustrated in figure 8.

Here we assumed that performed calculations have no deadline requirements but even then we could see that the simple approach into the analysis problem is not nearly adequate. Restricting the analysis into periodic events means the solution is still easily calculated, but accounting for aperiodic and sporadic events requires more extensive analysis. [5]

Possible feasibility analysis algorithms include for example:

- Rate Monotonic Analysis
- Deadline Monotonic Analysis
- Queue Analysis

8.1. Rate Monotonic Analysis

Rate Monotonic Analysis[7] was originally invented the beginning of '70s and later in '80s generalized enough to be practical. Original limitations of RMA were: [13, 10, 14]

- Fixed-priority scheduling
- Only periodical tasks
- Tasks do not synchronize
- Tasks do not suspend themselves
- Tasks can be instantly preempted

RMA is an analytic approach to predicting wether latency requirements set for the system are met. Execution and task parameters are used to calculate worst-case latencies for each thread and compared to the requirements set for the threads. Commonly revealed problems are priority inversions. [14]

The more general form of RMA is best suited for systems dominated by periodic or sporadic tasks for which time limits can be bounded without excessive variability. Systems in which WCET[8] are infrequent or there is no minimum interval between invocations are not best suited for RMA analysis. When dominating tasks are aperiodic, another analysis algorithm is needed. Although originally intended for off-line analysis, there are no reasons why RMA could not be applied on-line with modern computational capabilities. [14]

8.2. Deadline Monotonic Analysis

Deadline Monotonic Analysis[9] is a method similar to RMA and suffers from the same limitations. In DMA tasks that have deadlines sooner are assigned higher priorities and probability that deadlines are met is increased by that way. Further, it can be shown that if a set of periodic tasks is schedulable, it is schedulable by deadline monotonic algorithm.

[7] RMA
[8] Worst-Case Execution Time
[9] DMA

If we examine the 3 different periodic tasks in 1 we can show that they fulfill the requirement in 1.

$$\begin{aligned} \sum_{i=1}^{m} \frac{C_i}{P_i} &\leq 1 \\ \frac{C_a}{P_a} + \frac{C_b}{P_b} + \frac{C_c}{P_c} &\leq 1 \\ \frac{0,05}{0,3} + \frac{0,2}{0,5} + \frac{0,15}{0,4} &\leq 1 \end{aligned}$$

If we assume that all events a, b and c are simultaneously fired at time 0 rate monotonic scheduling fails to meet the deadlines. Events are handled by the order of their frequency, ie. a, c and b as last which fails to meet its deadline. With deadline monotonic scheduling the execution order is a, b and c as last and all deadlines are met.

8.3. Queue Analysis

Queue analysis is required for systems dominated by aperiodic tasks. The analysis derives directly from queue theory and is based on the fact that the queueing system must satisfy 2 where ρ is the server utilization, λ the number of events per time unit and μ the number of events possibly served in one time unit.

$$\rho = \lambda/\mu \leq 1 \tag{2}$$

9. Asynchrony

Often external events dictate execution times in RT systems, thus RTSJ defines mechanisms to bind execution of program logic to internal and external[10] events. Further RTSJ specifies methods for asynchronous transfer of control and thread termination. An event may be fired by application logic or an event external to the system, such as hardware interrupt. [8]

9.1. Asynchronous Events

Asynchronous events are handled by AsyncEvent which corresponds to the event and its probability of occurrence. Occurrence of the event is expressed by a call into fire() method. The fire() -method does not pass data for the handler, application needing to associate data with asynchronous events must do it by explicitly setting up a buffer. [8]

An object subclassing abstract class AsyncEventHandler or BoundAsyncEventHandler is scheduled in response to an occurrence of an event. BoundAsyncEventHandler is used when it must be ensured that each event is handled by a separate thread. When the event occurs, all event handlers bound to the event are scheduled for execution. [8]

[10] asynchronous

Table 1. Different tasks

Task	P_i	C_i	Deadline
a	0,3	0,05	0,1
b	0,5	0,2	0,25
c	0,4	0,15	0,4

9.2. Asynchronous Transfer of Control

Asynchronous Transfer of Control[11] is designed to allow efficient execution for a wide variety of RT systems. In RT systems there can be a large number of potential events with only a minimal number used at the same time, where dedicating a separate thread for each event seems excessive. RTSJ also allows program to bound an event to specific thread by BoundAsyncEventHandler. [8]

To provide better ATC RTSJ overloads the interrupt() method in java.lang.Thread offering non-polling execution control. It is based on throwing, deferring and propagating exceptions. ATC uses AsynchronouslyInterruptedException[12] for operation and methods must indicate this by including it in their throws-clause where applicable. [8]

Program may initiate ATC by throwing AIE itself and are subject to receive AIE if it has been declared to be thrown by the method. Exception to this is when execution of such method is in synchronized context, in which case AIE is put into pending state and received after execution leaves synchronized context. Also constructors are allowed to declare AIE as thrown and are thus interruptible. Thread subject to receive an AIE must respond to that exception within a bounded number of byte codes and the worst-case response time must be documented. [8]

If thread is interrupted in ATC method, AsynchronouslyInterruptedException is thrown once the thread is outside of a section in which ATC is deferred. Programmer may take advantage of this by using catch-clauses or the low- and high-level systems provided by RTSJ. [8]

Potential problem of catch-all clauses

```
try \{ ... \} catch (Throwable t) \{ ... \}
```

is worked around by requiring special schematics for catching AIEs. AIE is propagated forward unless handler calls happened() -method in the catch-clause. [8]

[11] ATC
[12] AIE

9.3. Asynchronous Thread Termination

Asynchronous thread termination works by the same mechanism as ATC, however threads are immortal as the termination would need to invoke methods that have AIE in theirs throws-clause. [8]

10. Conclusions

RTSJ seems a flexible enough framework to allow development of a wide variety of Real-Time systems and solutions. Minimal implementation of the virtual machine as required by the specification allows development of Real-Time systems only in limited scope. The framework is general enough to allow for development of a virtual machine tailored to the special needs of application and it could well be impossible to develope a general-purpose virtual machine that would suit all different applications.

References

[1] M. Barr and B. Frank. Java in Embedded Systems. *Embedded Systems Programming*, pages 24–32, May 1997.

[2] B. J. Brosgol. Ada and Java: Real-Time Advantages. *Embedded Systems Programming*, Nov 2003.

[3] B. M. Brosgol. Introduction to Real-Time Java. *Embedded Systems Programming*, Apr 2003.

[4] F. Bruin, F. Deladerriere, and F. Siebert. A Standard Java Virtual Machine for Real-Time Embedded Systems. *Proceedings of DASIA 2003 (ESA SP-532)*, 2003.

[5] A. Burns and A. Wellings. Processing group parameters in the real-time specification for java. In *On the Move to Meaningull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume LNCS 2889, pages 360–370. Springer, 2003.

[6] H. Cai and A. Wellings. Towards a high integrity real-time java virtual machine. In *On the Move to Meaningull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume LNCS 2889, pages 319–334. Springer, 2003.

[7] P. Dibble. *Real-Time Java Platform Programming*. The Sun Microsystems Press, 2002.

[8] G. B. et al. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[9] E. Y.-S. Hu, G. Bernat, and A. Wellings. A Static Timing Analysis Environment Using Java Achitecture for Safety Critical Real-Time Systems. *In Proceedings of 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS-2002)*, pages 64–71, Jan 2002.

[10] C. L. Liu and J. W. Layland. Scheduling algorithms for multi-programming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20:40–61, Jan 1973.

[11] R. Rajkumar, L. Sha, and S. Sathaye. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[12] M. Schoeberl. Restrictions of Java for Embedded Real-Time Systems. *In Proceedings of 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 93–100, May 2004.

[13] O. Serlin. Scheduling of time critical processes. In *Proceedings of the Spring Joint Computer Conference*, pages 925–932. American Federation of Information Processing Societies, 1972.

[14] Sha, Klein, and Goodenough. Rate monotonic analysis for real-time systems. In *Foundations of Real-Time Computing: Scheduling and Resource Management*, pages 129–155. Kluwer Academic Publishers, 1991.

[15] A. S. Tanenbaum. *Modern Operating System*. Prentice-Hall Inc, 2001.

[16] A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2003.

Introduction to Real-Time Programming with Ada 95

Juha Tukkinen
Helsinki University of Technology
Laboratory of Information Processing Science
P.O.Box 5400, FI-02015 HUT, Finland
juha.tukkinen@hut.fi

Abstract

Real-time programming is a very demanding task. The majority of current programming tools such as programming languages, operating systems and design tools are not suitable for real-time programming. When the system's correct behavior is also dependant on the timing of the computations, tools used must support this. This survey provides basic facts how Ada 95 can be used as a tool for real-time programming, instead of a tutorial of Ada 95 programming in general. This high level object-oriented language is still adequate for developing portable real-time applications. Having sophisticated built-in support for concurrency and real-time features, it is one of the very few realistic alternatives to C or assembler programming language in this domain.

1. Introduction

The correctness of a real-time system's behavior is dependant also on the timing of the computation results, not just the correctness of the results themselves. These systems must be predictable in execution time and react within certain deadlines.

A major problem in real-time programming is the large variety of tools. Real-time programs need support from the language (compiler), operating system and the hardware. The high level of abstraction that the tools must provide are, for example, scheduling and memory allocation.

Ada 95 is a language designed to be used in real-time programming with its extensions. It can make use of the concurrency features of real-time operating systems or executives. The main goals in Ada design have been high reliability, flexibility, reusability and maintainability in safety critical environments [5].

In real life, Ada 95 has been used in banking and financial systems, information systems, air traffic management systems, commercial aviation, commercial rockets, commercial imaging space vehicles, satellites, data communications, scientific space vehicles, railways, shipboard control systems, TV/entertainment industry, medical industry, general industry, military applications and by NASA. [4]

The structure of this article is as follows. The next chapter views the background and history of the Ada language with main focus on real-time programming features. Chapter 3 explains how the Ada 95 language supports concurrent programming. Chapter 4 describes how scheduling is done in real-time with Ada 95. Memory management is thought over in chapter 5. Chapter 6 covers different levels of communication. The last chapter concludes the article with a summary and a discussion of the possible future real-time features in Ada 0Y.

2 Background

Ada 83 had only preemptive scheduling based on static priorities. The problem of priority inversion was not solved. This means that threads with a lower priority might block higher-priority threads through the use of synchronization models. Also there was no model how task priorities interact with interrups and protected objects. The design was later on thought to be not enough portable and too restrictive.

There was an emerging need of support for dynamic priorities, better scheduling and tasking model design and more advanced timing features. The Ada 95 language extends Ada 83 by supporting these features and also for example abort-deferred regions and explicit synchronous and asynchronous task control protocols. [5]

In this article, real-time programming in Ada 95 is explained. The reader is presumed to have basic knowledge of the Ada language syntax and semantics. A good source to learn Ada from is for example *Ada Distilled* [8].

The core Ada language has no support for real-time features. However, some features, like concurrency support, not specific only to the real-time programming domain are

already built in the Ada 95 language. The Annex D in language reference manual [6] augments the core Ada language semantics by single solutions to problems in real-time programming.

Using these portable set of real-time capabilities features also poses a set of restrictions for the core language itself. These real-time features are invoked by using so called *configuration pragmas*. The semantics they correspond to are therefore only available to the one active partition of the program.

Next, the focus is in important features needed in real-time programming. Concurrent programming, scheduling, memory management and communication in task level and device level in Ada 95 are explained.

3 Concurrency support

Ada supports concurrency by multitasking and by distributed objects. Multitasking, called tasking in Ada, is implemented using only two types additional to the standard syntax and semantics of Ada. The semantics of these types, *task types* and *protected types* are augmented in Anneces C and D. Distributed programming is specified in the Annex E and it is not discussed in the scope of this article.

Because Ada has an integrated concurrency model, no external programming interfaces or specific operating systems support is needed. Concurrency is an essentially used feature in real-time programming, having a standard way to provide it improves portability.

An example program demonstrating both Ada concurrency features and its syntax in general can be seen in Figure 1. The trivial program runs two tasks, the first one prints out a sequence of numbers and the other one waits for user input to terminate the program.

3.1 Tasks

A task is an concurrent active object and it can be either in executing or suspended state. A task has a *base* and an *active* priority.

The base priority is given to a task when it is created. It can also be dynamically set later. The active priority of a task changes more often, although it reflects the base priority. The active priority changes usually by *priority inheritance*. It is used when the task competes for resources, it determines the task's position in the priority queue. To be precise, the active priority of a task is the maximum of the inherited priorities. A task inherits another task's base priority.

Critical regions for tasks are accomplished with *do* parts. Until this critical region is finished, the calling task is suspended.

3.2 Protected objects

Ada does not use mutexes or semaphores for concurrency. On the contrary, the *protected object* mechanism is used to allow the design of self-locking, encapsulated objects.

A protected object has protected components and operations. This means that the implementation must ensure mutually exclusive access to the object across tasks. An example of an protected object is shown in Figure 4.

4 Real-time scheduling

The real-time scheduler is responsible how the previously discussed components competing for resources are selected to be run. Compared to scheduling in more traditional environments, real-time schedulers need static timing information like possible execution time boundaries beforehand. Their behavior has to be very predictable and reliable.

Real-time scheduling is specified by the task dispatching model in Annex D of the Ada language reference manual. The tasks are selected to be run based on priority-ordered ready queues. The scheduling is *preemptive*. Details on the history of priority-based scheduling theory in Ada can be found from [9].

4.1 Task dispatching

The basic idea behind this is that a task becomes a running task only when the resources needed are available and the task itself is ready. Resources like processors are allocated based on the active priority of a task. The task dispatching takes place on so called *dispatching points*. These points occur, for example, when a task blocks, becomes ready or gets terminated or preempted.

Each processor has a ready queue, an ordered list of ready tasks, per a priority value. A task is in this queue only if it is ready or already running. A task can be on more than one processor's ready queue but it can be only running on one processor.

The only standard task dispatching policy that must be implemented is `FIFO_Within_Priorities`. The policy covers details not described in the task dispatching model like when and where the task is put in the ready queue after certain events. It is described in detail in chapter D.2.2 in [6].

4.2 Priority inversion problem

The situtation, when a lower priority task prevents a ready task with a higher priority to be run, is referred to as the priority inversion problem. This is a higly unwanted

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Task_Test is
  Time_To_Quit :  Boolean := False;
  pragma Atomic(Time_To_Quit);
  task Multiplier_Task;
  task Counter_Task;
  task User_Task;

  task body Counter_Task is
    Increment :  Integer := 1;
  begin
    while not Time_To_Quit loop
      Put("Counter task:  ");
      Put_Line(Integer'Image(Increment));
      Increment := Increment + 1;
      delay 0.6;
    end loop;
  end Counter_Task;

  task body User_Task is
    Char :  Character;
  begin
    loop
      Get(Char);
      Skip_Line;
      if Char = 'q' then
         Time_To_Quit := True;
         exit;
      end if;
    end loop;
  end User_Task;

-- main loop here
begin
  loop
    delay 1.0;
    exit when Time_To_Quit;
  end loop;
end Task_Test;
```

Figure 1. Example of an concurrent program in Ada 95

situation especially in the real-time domain as it can easily disturb the system's schedulability.

The priority inversion is the time a task remains at the head of the highest priority ready queue while a task that has a lower priority is executed. The time is implementation dependant and the maximum priority inversion must be documented by the Ada compiler specification.

As this is a big problem in real-time programming, Ada 95 programmer has means to minimize the risk of it by making use of protected objects and ceiling priorities.

4.3 Protected object's priority ceiling

Protected objects have a static priority ceiling. A locking policy describes the details how protected objects are locked in relation with tasks. The predefined locking policy `Ceiling_Locking` determines that the active priority of a task cannot be higher than the ceiling priority of a protected object, when the task calls protected operations from the protected object.

When a task executes a protected action, it inherits the ceiling priority of the protected object. A calling task cannot have a higher active priority than the ceiling priority of the called protected object. Failed check leads to raising an program error.

4.4 Real-time tasking specialities

A useful feature in some situations in real-time programming domain is the immediate task abortion. This is called *preemptive abort* in Ada 95. A task can preempt and abort another task if needed. The implementation must document the upper time it takes to abort and verify that a task has been terminated.

An efficient and simple capability to suspend another task instead of aborting is needed in some real-time applications. Asynchronous transfer of control (ATC) means that an schedulable object's point of execution may be changed by another object. An example can be seen in Figure 2.

4.5 Ravenscar profile

Tasking and other restrictions can be used in Ada 95 to prevent the use of unsafe language features. If the general run-time model of Ada would cause extensive costs or unsafety, a subset of its features can be taken into use with the pragma Restrictions.

By using the *Ravenscar profile* [1], applications can be guaranteed to be deterministic. They can be analyzed by static program analysis tools. When using this profile, for example the use of rendezvous in task communication (explained below) is prohibited.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Get_Name is
  Name :  String (1..80);
  Length :  Integer;
begin
  select
    delay 5.0;
    New_Line;
    Put_Line ("You are too slow.");
  then abort
    Put ("Enter your name in 5s:  ");
    Get_Line (Name, Length);
    Put ("Hello, ");
    Put (Name (1..Length));
  end select;
end Get_Name;
```

Figure 2. Delay statement and asynchronous transfer of control

4.6 Code predictability

As mentioned before, real-time scheduler needs static timing information to operate correctly. This is highly essential in the hard real-time environments. Worst case execution time (WCET)-analysable sporadic hard real-time tasks can be implemented in Ada as described in [2]. This is done run-time by using Deadline Monotonic Scheduling Analysis instead of too heavy Rate Monotonic Analysis. WCET calculation has also been researched e.g. in [3].

In general, elapsed time measurement and precise periodic execution are estabilished by `Real_Time.Time` and delay statement. Also, certain restrictions must be taken in to use that are described later on.

5 Memory management

Ada 95 does not provide automatic garbage collection for the programmer. However, compilers may provide an implementation of this feature. As this will cause unpredictability in the execution time and space, it is not very usable in the real-time programming domain.

The memory can be dynamically reserved at with a reserved word *new*. It is called an *allocator* and it reserves memory at run time for an object of some type. Pointer arithmetics are by default prohibited in Ada contrary to, for example, C or C++. This is a partial result from the requirement to minimize side effects in the language. Dynamically allocated areas of memory are handled by the package `System.Storage_Pools` that is described in Chapter 13 of [7].

Ada run-time executives usually map into the underlying operating systems on memory management. Ada 95 can make use of the underlying real-time operating system (RTOS). It is implementation dependant, what can be done real-time. Usually, hard real-time programs need to be very deterministic on memory usage and there is little need for large scale dynamic memory allocation and deallocation.

6 Communication

In this chapter, communication is divided into two levels. First, inter-process communication in the Ada 95 language is described. Next, more general level of interaction with the real world is described. Input/output is a critical part in real-time environments as this is the part where the real-time programming links with the real world.

6.1 Inter-process communication

Tasks in Ada communicate with each other usually with *rendezvous*. A task that wishes to communicate with another task places a request for rendezvous in the *entry queue* of the called task. Then the calling task is suspended until the request for this anonymous rendezvous, or entry, in the queue is consumed and the called task has processed this entry. However, the calling task may as a part of the call ask that the request will remain in the queue only for a limited period of time.

Additional to the direct communication mechanism rendezvous, indirect communication is possible by using

```
pragma Task_Dispatching_Policy(
  FIFO_Within_Priorities);
Package Position is
  Sens_Int_Lvl :  constant := 15;

protected Value is
  procedure Sensor;
  pragma Interrupt_Priority(Sens_Int_Lvl);
  pragma Interrupt_Handler(Sensor);
  pragma Attach_Handler(Sensor,
   Sens_Int_Lvl);
  procedure Get_Value(Value :
    out Interfaces.Integer_32);
  private
    Value :  Interfaces.Integer_32;
end Value;
end Position;
```

Figure 3. Example of an protected object and an interrupt handler

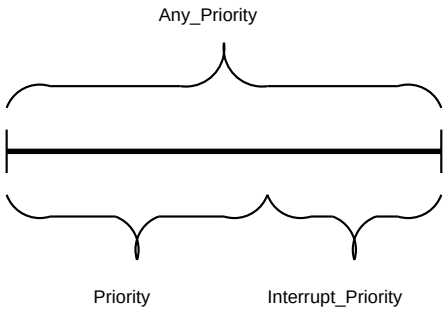


Figure 4. Ada priorities

shared data or protected objects. Rendezvous has been thought to be too complicated to be properly statically analyzed and is prohibited in some language restrictions as described above.

6.2 Input/Output

The Systems Programming Annex (Annex C) provides many low-level programming features. For example, it is possible to access memory directly by an address and interrupt handlers can be programmed. An code sketch of the latter can be seen in Figure 3.

The package `Ada.Real_Time` provides that accurate time sources can be used. A task may delay itself accurately as was seen in Figure 2. A single time unit must be less than 20 microseconds. Numerous of details of time are considered to be documented, such as the upper bound on clock drift rate.

The different interrupt subtypes can bee seen in Figure 4. The `Interrupt_Priority` is typically higher than a `Priority` of a task so that a task would not block an interrupt. The minimum level of interrupts that need to be supported by the hardware is 31.

The low-level programming is not so dangerous as it is in, for example, C. There are no true pointers in Ada. A corresponding capability is achieved with *access types*. They are safe by default, there are no void pointers.

Programming at the machine level is possible by deal with addresses and raw storage as shown in example in Figure 5. Also, Ada 95 programs can interface with native code, if necessary. Subprograms can be written that have inlined assembler code with the package `System.Machine_Code`.

7. Conclusions

This article has given a brief introduction what real-time programming is like using Ada 95. It can be said that Ada 95 is a language suitable for real-time programming.

```
Read_Value :  Interfaces.Integer_32;
for Read_Value'Address use
  System.Storage_Elements.
  To_Address(16#1024);
```

Figure 5. Reading an integer from a memory address

The built-in facilities to support real-time programming are quite rare among other languages. Ada was designed to promote sound software engineering practices from the beginning. It has a high performance and an encapsulated tasking model.

The descrease of popularity seems to be the biggest risk in using Ada 95. It is much more harder to find skilled Ada programmers than, for example, adept C staff. Therefore, it is also increasingly hard to find supported compilers and platforms to be used in real-time programming. The future of the language itself does not seem to depend critically on the newcoming reversion of the language, Ada 0Y.

References

[1] A. Burns. The ravenscar profile. *Ada Lett.*, XIX(4):49–52, 1999.

[2] A. Burns and A. J. Wellings. Implementing analysable hard real-time sporadic tasks in Ada 9X. *ACM SIGADA Ada Letters*, 14(1):38–49, /1994.

[3] R. Chapman, A. Burns, and A. Wellings. Static worst-case timing analysis of ada. *Ada Lett.*, XIV(5):88–91, 1994.

[4] M. B. Feldman. Who's Using Ada? Real-World Projects Powered by the Ada Programming Language, August 2002. Web page. Referenced in October 2004. `http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html`.

[5] International Standard ISO/IEC 8652:1995(E) with COR.1:2000. *Ada 95 Rationale: The Language, the Standard Libraries*, 1997. Lecture Notes in Computer Science, 1247.

[6] International Standard ISO/IEC 8652:1995(E) with COR.1:2000. *Ada95 Language Reference Manual*, 2000. Version 6.0.

[7] International Standard ISO/IEC 8652:1995(E) with COR.1:2000. *Annotated Ada95 Language Reference Manual: Language and Standard Libraries*, 2000. The MITRE Corporation, Inc.

[8] R. Riehle. *Ada Distilled: An Introduction to Ada Programming for Experienced Computer Programmers*. AdaWorks Software Engineering, 2003.

[9] L. Sha and J. B. Goodenough. Real-Time Scheduling Theory and Ada. *Computer*, 23(4):53–62, 1990.

Using Rational Rose and Rational Rose RT in Real-Time Development

Timo Toivanen
Helsinki University of Technology
Computer Science Laboratory
Timo.Toivanen@iki.fi

Abstract

Using modeling software and automatic code generators instead of writing code from scratch has been an option for some time at least in technically less demanding programs but now it is taking foothold also in real-time development. Using generated code will help to avoid many nasty caveats and it will speed up the development process. UML-based code generators do still have problems in real-time domain, but the solution exists. This article will give a short introduction to two code generation tools and describe some of concepts related to these tools.

1 Introduction

Almost always when designing a software there are at least some architecture diagrams involved. Sometimes these diagrams are so accurate and extensive that they can be converted to code to be used as a template for programmers or even to a complete runnable program. The notation of these diagrams must be well defined before any conversion can be done. There are few standardized notations for architectural diagrams of which UML[1] is the de facto standard in object oriented design. Other notations are not so widely used but they have their own application areas. For example SDL[2] is standard in telecommunication area because its well suited to describe protocols.

Rational Software has developed a toolkit to draw UML diagrams and to generate code from these diagrams. Rational Rose is able to produce skeleton code of the program's class structure. Rose's extended version, Rational Rose RealTime uses real-time extensions of the UML and is able to generate a complete, runnable program.[3]

Rose family is not the only toolkit to generate programs graphically. Open source project Poseidon is mainly an UML-editor but it can also generate skeleton code.[1]

Simulink is widely used on developing simulation programs and it's approach is a bit different. It uses small blocks of code that may come from simulink's libraries or be hand written. These blocks are wired in series or parallel to form a useful program. [5]

This article concentrates on UML and Rose family.

2 Why to use these tools

Program's architecture is easier to understand an comprehend when its represented graphically in well defined format. This also helps to avoid problems and bugs caused by interpretation errors between modeling and implementation teams. In architectural level these interpretation errors are completely avoided when at least skeleton code is automatically generated from design models.

With appropriate tools the entire code can be generated from the design model. This raises the abstraction level of the development process since most of the "coding" is done graphically and only the hardware related parts must be coded manually. This eliminates significant number of bugs that usually result from programmer's carelessness. The automation also speeds up the development process.

3 Standard UML

Parts of the UML notation can be divided to three categories, elements that are basic building blocks of the language, relationships between these blocks and finally diagrams they form together. From this article's point of view the most important parts are class diagrams, state diagrams and the elements and relationships needed to build these diagrams.[8]

Class diagram models the program's internal structure in object oriented design. It shows what classes the program has and how they are related to each other. These relations can be for example generalizations (means inheritance), aggregations, compositions or some one of the more rarely used types.[8]

[1] Unified Modeling Language
[2] Specification and Description Language

Interaction between classes can be described with ether sequence diagrams or collaboration diagrams. The sequence diagram emphasize time while the collaboration diagram shows the message paths between classes. [8]

State diagrams are important here because they are the UML's only means to describe functionality in such form that it can be converted to working code. A basic state diagram contains states and transitions between them. Self transitions are also allowed.[8]

The rest of the UML's diagrams are omitted since they are not important in this context.

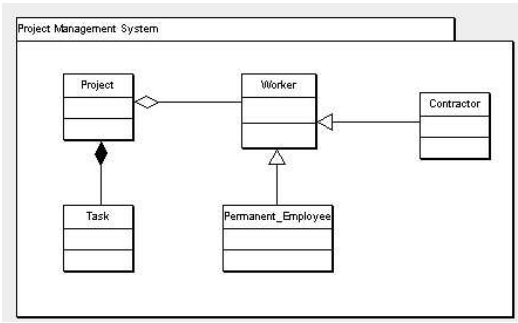


Figure 1. An example of class diagram in standard UML

4 UML real time extensions

The basic version of the UML cannot sufficiently describe complex real time systems so few additions are needed. They are not included in basic UML to keep the language itself simple. UML for RealTime is actually a combination of the standard UML and RealTime Object Oriented Modeling language (ROOM) developed by ObjecTime Limited.[4]

4.1 Capsules

Capsule is a specialized version of an ordinary class. They can participate in many relationships specified in standard UML and they can contain attributes or operations. The major difference is that capsules cannot have public operations or attributes and they can use ports to communicate. From this article's point of view the the most important feature of capsule is its internal state machine. This machine is described by UML state diagram and it gives the capsule its functionality. [8]

4.2 Ports

A port is a communication channel the capsule uses to interact with other capsules. The capsule's methods nor fields are accessible from the outside so ports are the only means of communication. A port can be public or protected, protected ports are used for example in timing. [8]

4.3 Protocols

Entities the capsules send to each other are called messages. A message may or may not contain data depending on its use. A message can transport a primitive type data or an objects. Messages without data are used only to trigger state transitions inside the capsule. A protocol is a set of messages which two or more capsules can use to communicate. [8]

4.4 Diagrams

Relations of capsules and protocols are defined in class diagrams. Capsules can be generalized on associated just as classes and associations can occur even between classes and capsules. A protocol is connected with aggregations to every capsule that uses it.

Message paths between capsules are defined in a capsule structure diagram that resembles a collaboration diagram. Message paths are defined by wiring capsules' ports together.[8]

5 Rational tools

5.1 Basic Rational Rose

Rational Rose is a graphical modeling tool-set to create UML-diagrams. Currently Rose is much more than a diagram drawing tool and one of its most interesting features is the capability to generate skeleton code from the UML-model's class diagram. This code contains class definitions, attribute and method declarations and references between classes generated from relationships. This code is still far from runnable program and its intended as a template for coders. Rose is also able to read source code and generate standard UML diagrams from it.

5.2 Rational Rose RealTime

Rational Rose RealTime is a visual UML-modeler combined with code generator capable of producing runnable code and graphical debugger. RoseRT emerged when ObjecTime Developer was integrated to Rational Rose. With

a suitable compiler RoseRT forms a complete development environment. RoseRT uses UML-RT notation for modeling. A program generated with RoseRT runs at top of the service library that provides message handling and schedules execution between capsules if they are running in the same thread.[2, 6]

5.2.1 Defining the structure

When modeling a program with RoseRT, the capsule is the basic building block. It represents an autonomous object that has it's own properties and and behavior. Capsules and their relationships to other capsules and protocols are defined in a class diagram as stated in UML specification. One of the capsules is selected as a top capsule which "owns" other capsules. This "owning" is denoted with aggregations and the top capsule will contain the main method when code is generated. The class diagram can also contain ordinary classes that can be used for example to transfer data inside messages.

As in UML-RT specification, the structure diagram shows how the capsules are wired together. Ports are used as junction points and one port can connect only one other port, so a capsule needs to have an own port for each capsule it wants to communicate with.[8, 9] Wirings and ports can be created manually or by using "Aggregation tool" that creates them.

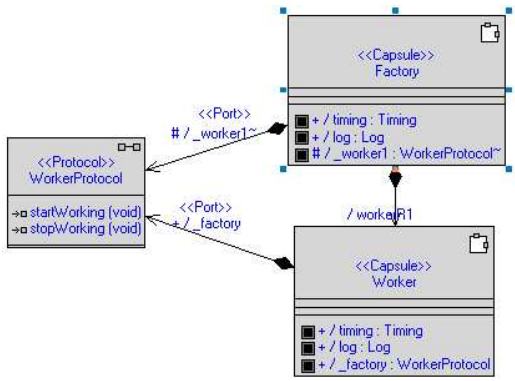


Figure 2. An example of class diagram containing capsules

5.2.2 Defining the functionality

All of the program's functionality is in state diagrams inside it's capsules. State transitions are triggered with messages coming form the capsule's ports. If timed transitions are needed, a special *timing*-port can me added to the capsule and then transitions can be set to trigger at a certain moment of time. Some parts of the program (e.g. sending a message or using external functions) must still be coded so there are three places to enter code in state diagram. *Action code* is executed when arriving to the state and *Exit code* is executed when leaving it. Also state transitions can contain lines of source code.[8, 9]

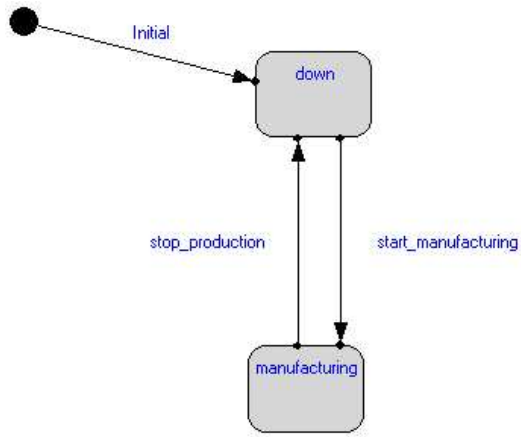


Figure 3. An example of state diagram that defines the functionality of an capsule

5.2.3 How the code is generated

In theory the model is converted to source code by using few basic methods. Each capsule convert to a class in final code. State diagrams convert to state machines that are constructed of case-statements that are nested if necessary. Each capsule can also form an own thread or they all can run in the same thread depending the capabilities of the OS in use. In real life RoseRT generates quite complex code an the process itself is largely undisclosed.

5.2.4 Graphical debugging

RoseRT's graphical debugger is very useful since the programmer can run the program step by step and see how the states transit in the diagrams. Also messages can be injected manually to a running program. RoseRT can create state diagrams from test runs so message traffic between capsules is easy to verify.[9]

5.2.5 Real time features

RoseRT supports dozens of real time operating systems like QNX and RTLinux. It also has a accurate clock that can be used to trigger state transitions at the specific moment

of time. However, RoseRT does not have any means to define execution deadlines even though it is claimed to be a "real-time" developing tool. Nether it does not support use of third party analysis tools like TimeWiz or RapidRMA. If hard real-time is needed, it is left to the programmer to make sure that deadlines are met. One must manually fit the program structure to the thread model of the RTOS in use. Also the scheduling discipline must chosen so that constraints are satisfied.[2, 6] One reason for this might be that ROOM and UML-RT are mainly targeted to telecommunication area where deadlines are more often soft than hard.[2]

Though RoseRT does not support it, there is a way to perform a schedulability analysis UML-RT models. This analysis method developed by M. Saksena and P. Karvelas is usable for both single-thread and multi-thread programs. [6] Further discussion of this method is beyond of the scope of this article.

6 Case 1: modeling process using Rational Rose RT

When developing a simple program the process is quite simple and it goes as follows:[9]

- Create all necessary capsules and select one of them to a top capsule
- Create necessary ports to the capsules
- Create necessary protocols and wire capsules together or use aggregation tool
- Draw state diagrams that define the functionality of the capsules
- Add transition triggers and action codes to state transitions of the state diagrams
- Build, compile and run the model
- If the result is not the one desired, fix the model (use graphical debugger if necessary) and try it again

Program can also be developed one feature at the time when it can be tested after each feature. This method is advertised by Rational and it is called Rational Unified Process (RUP) and it is very well suited to used with automatic code generation tools.[8]

7 Case 2: Generating Ada-code from UML model with Rational Rose

As mentioned earlier, also the basic version of Rational Rose has some code generation capabilities. Rose has support for both Ada 83 and Ada 95 and the generated code has some differences depending the used Ada-version. Rose can convert the following parts of the UML-model to Ada-code:

- Classes with their field and method declarations
- Every method declaration generates also a subprogram declaration
- "has"-relationships of the classes
- Navigable associations convert to components
- In Ada 95 the generalization converts to a type derivation, but in Ada 83 it converts to component and all inherited methods are duplicated
- User-defined operations can be converted stubs if desired

Rose can also function in backwards and perform reverse engineering. An UML model can be generated from source code and even from compiled program, if Rational's Apex-compiler is used. Two types of diagrams can be generated: Class diagrams and component diagrams that can be ether unit diagrams or subsystem diagrams. [7].

8 Conclusions

Interest in using object oriented design is growing all the time also in real-time domain and now there are tools like RoseRT which allow programmers to generate fully working and runnable code directly from UML-models. Automatic code generation provides many advantages compared to conventional development and makes them very enticing.

The problem is that these tools, even though they claim to be "real-time" development tools, do not have means to make sure that real-time constraints are met. However, there are methods to perform schedulability analysis on UML-RT models, but the results depend heavily on underlying OS and the development tools do not yet support the analysis. Automatic code generation can be used in hard real-time environment, but it is up to the programmer to worry about the demands of the domain.

References

[1] Gentleware. Poseidon for uml. http://www.gentleware.com, 2004.

[2] Z. Gu and K. G. Shin. Synthesis of real-time implementations from uml-rt models. In *RTAS Workshop on Model-Driven Embedded Systems (MoDES 2004)*, Toronto, Canada, 2004.

[3] IBM. Ibm rational software website. http://www-306.ibm.com/software/rational/. Referenced 1.10.2004.

[4] A. Lyons. Uml for real-time overview, April 1998.

[5] T. Mathworks. Simulink - simulation and model-based design. http://www.mathworks.com/products/simulink/. Referenced 1.10.2004.

[6] M. Saksena and P. Karvelas. Designing for schedulability: integrating schedulability analysis with object-oriented design. In *Proc.IEEE Euro-Micro Conference on Real-Time Systems*, pages 101–108. IEEE, 2000.

[7] R. software corporation. Using rose ada for forward and reverse engineering, 2001.

[8] R. software corporation. Modeling language guide, June 2002.

[9] R. software corporation. Toolset guide, 2002.

Introduction to Statecharts in Design of Reactive Systems

Mikko Byckling
Helsinki University of Technology
mbycklin@cc.hut.fi

Abstract

Reactive systems, i.e. systems which have to react to one or many external or internal impulses which arrive concurrently in near arbitrary order, are often difficult to develop. Most of the complexity in such a system roots to the multitude of different event combinations which may occur during its lifetime. Thus a method with a native support for parallellism is preferable. As many reactive systems are also safety critical, a formally verifiable design is needed.

This paper presents the use of statecharts as a design method for reactive systems. Reader is introduced to the original syntax of statecharts as presented by Harel in [4] and to some of their basic properties. As a syntactic example a simple case study of a burglar alarm system is presented. A design tool which uses statechart formalism, STATEMATE, is also briefly visited.

1 Introduction

Traditional computer systems accept one input, process it, and produce some kind of an output. These systems could also be called as *transformational* [7] and their development, in spite of its obvious problems, is reasonably well understood [17].

Other type of computer systems, which are actually more common than the previous but less observed, are *reactive* systems. They are largely event-driven and reacting to external and internal stimuli [7, 4]. Events handled by system may or may not depend on each other. For example, a system controlling a jet fighter missile system may depend on radar readings, but be independent on the position of pilots seat ejection handle.

Edwards [2] and Harel [7] define several types of reactive systems. These include vehicle control, digital watches, robots etc. Some of these systems may be used in life critical situations, i.e. failure in system may lead to injury or loss of human life. Therefore, ad-hoc approach to development of software is not acceptable. Formal methods must be used for verification to guarantee safety [2].

The problem of describing a reactive real-time system roots mainly to the concurrence of events. Many inputs may occur simultaneously and at an arbitrary order which creates a challenge in the definition of such systems, as different levels of abstraction are difficult or impossible to use.

Reactivity also causes a system to be dependent on time. Responses to events may have deadlines and missing such a deadline may, in the worst case, be equal to no response at all. As many events with different deadlines may occur simultaneously, system must be able to handle the worst case of such a combination to function correctly.

Several methods for successful development of reactive systems have been proposed and some of them, namely Structured design for real-time systems, Object oriented design (OOD), Process abstraction method for Embedded Large Applications (PAMELA) and Software Cost Reduction (SCR) have been compared by Kelly in [11]. Comparison in [11] is quite thorough and compares for example formality, semantics and validity of different techniques clearly.

This paper introduces reader to one design method for reactive real-time systems, the statecharts. Chapter two introduces technique for functional analysis of complex systems. Chapters three and four present the syntax and some semantics of statecharts and a short example of their use. Some properties of statecharts are discussed in chapter five. Chapter six introduces a product development tool, STATEMATE, which uses statecharts. Paper is concluded in chapter seven.

2 Structured systems analysis

For human beings large entities are often difficult to comprehend. It is a known fact, that understanding of constructs becomes easier via proper abstraction. Usually this abstraction takes advantage of the structure of the system or construct under development. Formed larger structures may be again divided to smaller pieces and thus the design becomes manageable.

Structured systems analysis (SSA) is a way of determining the structure of a system in an analytic way [11, 17].

This analysis, which may be used as a tool for obtaining a *functional decomposition* for the system, is a top-down method in which one starts analysing data flow by drawing a data flow diagram (DFD). Initial DFD is then refined stepwise to improve the functional view of the system.

There exists several notations for DFD [17]. The main point is to represent flow of data in the system. Data sources and destinations as well as processes that transform data and store it may be defined. This, after adequate refinement, results in functional view of the system, which may then be used to extract functions or actions for implementation phase. Extracted actions and flow of information may be hierarchic.

Structured systems analysis usually results in a good understanding of the systems behaviour. Unfortunately mistakes in the design phase may not show up until very late in the development [11]. SSA includes further steps not mentioned here. For a complete description see [11, 17].

It is worth noting that originally structured design was developed for transformational systems and has been further extended to be applicable for real-time. Thus the notion of time is not innate for SSA and requires some adaptation. Some temporal aspects of methods based on structured analysis are discussed on [16].

3 Statecharts

Statecharts are a formal definition language to confront the main problems in reactive real-time systems. They natively support hierarchy of objects, concurrence and general transitions between states. Global communication of objects is also possible. Statecharts try to address the difficult problem of moving from system definition to implementation by introducing *executable specifications* [4], i.e. specifications which may be used to generate executable code.

Originally statecharts were presented by Harel in [4]. Statecharts are a formal description especially suitable for reactive real-time systems. Many extensions exist, some of which were originally presented by Harel himself. Here reader is familiarized with the original syntax, according to Harel in [4]. References concerning temporal extensions to statecharts are, for example, [14] and [12].

Statecharts have been quite recently adopted also to Unified Modelling Language (UML) [15]. UML's *State Machines* are an object-based variant of Harels original statecharts. Largest difference between the two is that object statecharts define the behaviour of a type, rather than specifying the behaviour of processes. For a more comprehensive list of differences reader is adviced to consult UMLs semantic documentation [15].

One of the many good features of statecharts is the use of formal model to generate executable code. This option eliminates, or at least greatly reduces, the work needed for implementation of defined system. Thus the system may be defined at a more abstract level, alleviating the difficulty in realization. It also makes it easier to analyze possible defects in implemented system more efficiently, striking out the possibility of an implementation error.

3.1 States and state transitions

One could point out that statecharts are an extension to FSM formalism. A major difference is however, that statecharts actually take advantage of the *area* of the diagram, i.e. states may be grouped together inside a superstate. Syntactically, states are denoted as rounded rectangles and state transitions as arrows.

State transition is tied to certain event α. In statecharts this event may be optional by parenthesized condition C and trigger a slash-separated action β. Generally the conditions for state transitions may be written as

$$\alpha(C)/\beta \tag{1}$$

and as in conventional FSM formalism the system will end up in the state at the end of state transition arrow.

3.2 Hierarchical decompositions

Statescharts may express hierarchy of states by *encapsulation*. This means that if some system must be either in A or B, but not in both, states may be combined to a new superstate D. Semantically this relationship is for D an exclusive-or (XOR) relationship.

As for arrows leaving and entering D same type of abstraction may be used. In a more abstract level all arrows may be marked to enter state D. At a more refined level these arrows may actually be entrances to states A or B, or to a superstate D generally. Same syntax holds for arrows leaving D, i.e system may leave D if in A and event α occurs or leave if event β occurs unconditionally of internal state in D. This property of leaving a superstate, which basically means leaving all substates, is one of the main factors, by which statecharts economize the number of arrows.

Any level in statecharts may have a *default* state. This means that if system is asked to enter superstate D, default state of D is entered. This is marked as a small arrow beginning from a dot. Each state in hierarchy may have its own default state. Default states are actually similar to start states of FSMs, but may occur at various abstraction levels of statecharts. Hierarchical state formalism with default states is illustrated in figure 1.

3.3 Orthogonal decompositions

To express concurrency in statecharts AND decompositions are used. The term orthogonal derives from the prop-

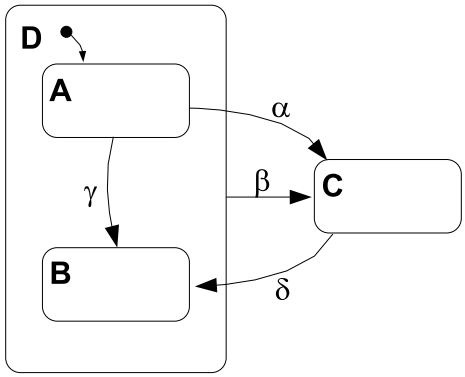


Figure 1. Hierarchical decomposition of states

erty, that different components of the systems AND decomposition are not aware, at least fully, of each other. Orthogonality ensures, that system development process may be simplified by isolating components not dependent on each other and developing these isolated components as separate entities.

Concurrent states exist simultaneously in the system, i.e. to be in some state the system must be in all of its AND components concurrently. Some superstate Z is therefore an orthogonal product of its substates. The notation is the splitting of some superstate into physically different components using dashed lines.

In figure 2, to be in Z means that system is in some combination of B or C and E or F. As before Z may be entered via default arrows or by direct arrows to any combination of its substates. Actually Z always is in some combination of states (s_A, s_D), where s_A and s_D are substates of A and D respectively.

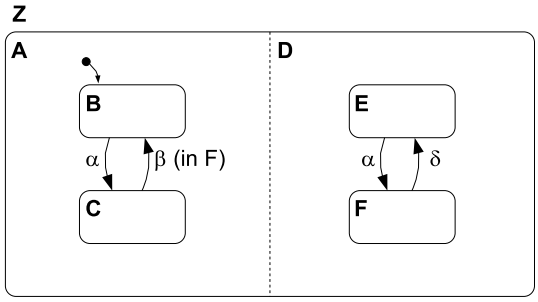


Figure 2. Orthogonal decomposition of states

Events may trigger state transitions in either or both of the orthogonal components. These transitions take place simultaneously in both components resulting in a new state combination $(\tilde{s}_A, \tilde{s}_D)$. The independence of states results in the fact that state transitions in Z do not depend on the state of other components of Z. As before orthogonal parts may have their respective default states which are used if superstate Z is entered. Additionally any combination of substates may be entered and left.

The power of orthogonal decompositions is in the number of states they are able to represent. In conventional FSM approach the number of states for a combined automaton is the cartesian product of its parts. This means that the number of states in the new automaton is the product of original number states in each automaton. This results in an exponential blow-up in the number of states. In statecharts however, no new states are required if the factors are orthogonal and thus this exponential increase in the number of states is avoided.

If some dependence between different states of a superstate is needed, that is if components are only partially orthogonal, a special "in S" condition may be used for state transitions. This condition causes a component to be partially dependent of some orthogonal component and state transition only takes place if that given component is in S. Another way to introduce subordination between orthogonal components is to use common events, i.e. events that trigger state transitions several orthogonal components.

3.4 State history

Statecharts further extend the conventional FSM formalism by introducing a new way of entering a group of states: entering the state most recently visited. This gives superstates the ability to memorize their last visited substate.

State history is denoted as circled **H** in a group of states. History may be applied all the way down to the lowest level by attaching an asterisk to the history entry, i.e. writing $\mathbf{H}^*$. It is possible to override state default entrances with history entrances by drawing a small arrow from default entrance dot to history connective and onwards to wanted default entry.

When a state is entered by history the most recently visited state is chosen. In the case of state decompositions, only one of the hierarchical substates is entered via state history. The defaults of decomposed substates are used if history is not applied down to lowest level and substates do not hold their own history entrances. Same applies to orthogonal states which have state history. Some or all orthogonal states may be entered via history entrance.

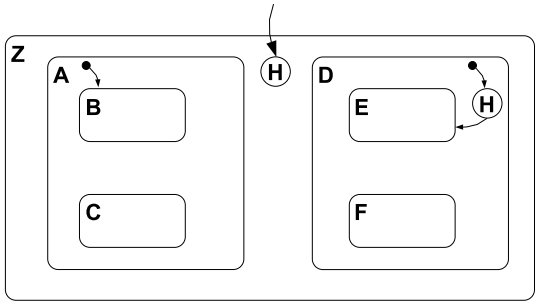


Figure 3. State history

3.5 Condition and selection entrances

Statecharts offer also other methods to more complicated entrances to substates of a superstate. If some event α depending on different conditions P, Q and R transfer system into substates A, B and C of D, then multiple input arrows may be replaced by a single arrow to a circled connective **C**, which stands for *conditional*, and further state transitions handled internally according to conditions P, Q, R in D. When same event conditionally chooses between different substates, enables condition entrance the use of single arrow to enter a superstate.

From the viewpoint of FSMs, a simple selection operation followed by an update requires quite complex design, if other operations are permitted between selection and update. Circled connective **S**, which stands for *selection*, may be used in such occasions. This is based on the fact that update event is actually the selection of one of defined states and an action related to it. Because usually selections of such an update are separate and only one item is selected, the use of such an entrance becomes plausible. Figure 4 describes the syntax of these entrances in pictorial form. Note the use of state history entry for update operation.

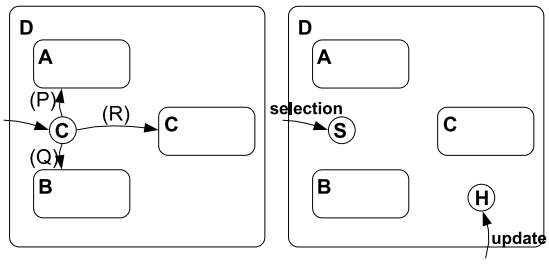


Figure 4. Condition and selection entrances

3.6 Delays and timeouts

Time is often an important factor in reactive systems. Some events may require that they are executed within a certain timeframe or after a certain specified time has passed. In original statechart formalism timeouts may be associated with a states and state transitions.

To give some temporal bounds for presense in a state one include a squiggle to upper left corner of state rounded rectangle to indicate that state comes with a bound. Lower and upper bounds on the time in a state are given with time limits attached to squiggle in form $\Delta t_1 < \Delta t_2$, where Δ_i are upper and lower bounds respectively. Either of these bounds may be omitted. Special state transition **timeout** handles state transition in the case timeout or delay takes place. This syntax is illustrated in figure 5

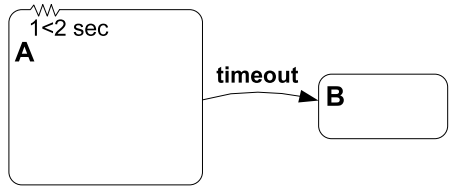


Figure 5. Delay and a timeout

3.7 Actions and activities

Statecharts use actions to connect state models to the so called "real" world. The ability to generate events and change the value of conditions in a statechart model has been made possible with the introduction of actions. Actions are the right side of slash separated syntax (1) attached to state transitions. The term *action* is used to express instantaneous occurrences the take zero time. It is worthwhile noting, that action β may also be an event, which is then sensed elsewhere in the system.

Closely related to actions are *activities*, which take some nonzero amount of time, and are thus not to be mixed with actions. In statecharts some special actions used to control activities are *start(X)* and *stop(X)*, where X is an activity. Additionally condition *active(X)* may also used.

Actions and activities require some understanding of the system under development. More precisely decomposition into subsystems and the functions and activities these subsystems support. This activity hierarchy is thus closely related to functional decomposition of the system, which is more closely studied in section 2.

In statecharts actions and activities are allowed also in states in addition to state transitions. This allows one to define activies that are active when system in a certain state.

One may define actions which are executed upon *entry* to, *throughout*, and *exit* from a state. Additionally linking actions to states takes into account the state hierarchy as shown in figure 6 and thus state transitions inside a superstate do no cause entry or exit actions of that superstate to take place.

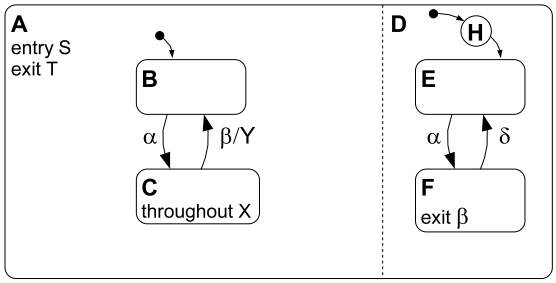


Figure 6. Actions and activities

4 Case study: A burglar alarm system

This section gives a short example about the use of statecharts for system design. A simple fictional burglar alarm system is defined with statechart syntax presented. The purpose of this example is to illustrate the ease of use and efficiency of statecharts compared to usual finite state machines. In this example different states of system are marked with **bold** and events in *italic* text.

4.1 Requirements

Let our model system be a simple burglar alarm system with three physical parts: sensors, alarm bell and indicator light. Our sensors have the purpose of triggering an alarm in the system. Additionally let our viewpoint not to be restricted by how the sensors work (using motion detection, pressure sensors etc.) or are there multiple sensor units or just one. Alarm bell has the usual function of making a loud noise when activated. A flashing indicator light, which is placed inside the guards office, has the function of giving visual stimulus.

Alarms raised by system may be silent or loud. A loud alarm, which is the default, consist of ringing the bell and flashing the light. Silent alarm is given when the bell has been muted by using a switch. Therefore during a silent alarm only the indicator light is flashed. This enables the guard to call the police before trying to catch the burglars. Bell may also be switched to muted state by user after alarm has been triggered. Burglar alarm may also be switched off during office hours. When switched off, settings of the system are reset to their defaults.

4.2 Statechart model

In this simple case, it is easy to identify different components of the system. When looking at the whole system it is easy to identify two basic states of operation: **On** and **Off**. Because switching the system on brings the system to its default state, powering off the system clears the **on** -state history. This is abbreviated as *clh(**on***)* -action in the *power off* -event of statechart.

The **On** -state may be further divided into components. When system is in its **On** -state, it obviously has a **mode** and **alarm** -states, which are partially independent of each other.

Mode -state is quite straightforward. There exists two internal states for system mode as stated by the requirements: **loud** and **silent**. The fact that system history has a memory is taken into account by attaching a history entry to **mode** -state, even though in this limited case history entrance is not needed. Events *mute* and *unmute* change the system mode from **loud** to **silent** and vice versa.

When system is on, the alarm may be active or deactive. Therefore **Alarm** -state was divided into **deactive** and **active** substates. Events *activate* and *deactivate* switch the alarm on and off respectively.

Active -state has two orthogonal substates: **bell** and **light** representing the part of the alarm system. **Light** -state is simple. When alarm is switched on, indicator light is flashed. This is being done by starting and stopping a flash light -activity when state is entered and left. **Bell** -state is aware of the system mode. If mode is loud, **ringing** -substate is entered. When system is in silent mode, bell is in **muted** state. This is done by attaching a condition entrance to **bell** -substate. Further changes in the system mode affect also the bell state, so therefore events *mute* and *unmute* change also the internal state of **bell**.

The resulting statechart for this system is given in figure 7. Given the simplicity of the system DFD was not used. Actually, in this case, if one starts to compose DFD according to [11, 17] the decomposition is probably quite like the given division of the system.

5 Properties of statechart formalism

The extensions statecharts provide, compared to conventional finite state machines, enable the use of formal models for the description of complex reactive systems. The most important factors in the way statecharts economize the number of arrows are state clustering and arrows with common sources or targets [4]. Thus it is possible to enter a superstate or orthogonal set of states with a minimal number of arrows via default substate entries and state histories.

Orthogonality enables statecharts to avoid exponential blowup in the number of states [4], which is the main prob-

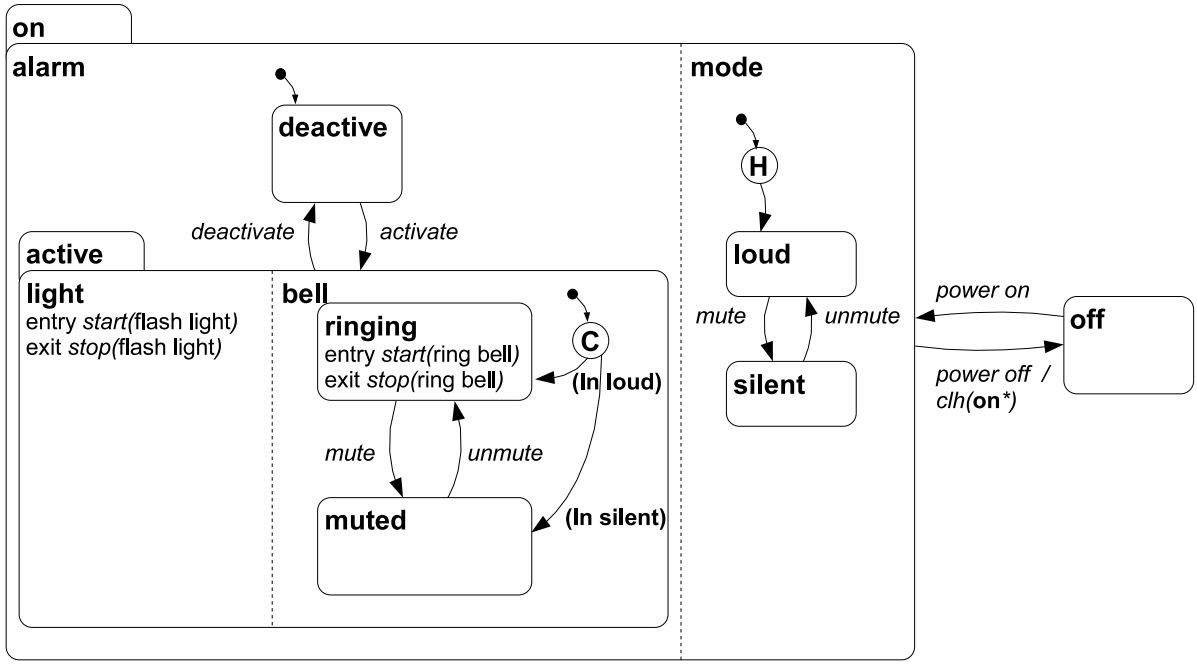


Figure 7. Statechart of the burglar alarm system

lem of FSMs applied to large constructs. Orthogonal states also simplify the design process by introducing the independence of different parts of the model.

From a more theoretical point of view, statecharts have a triple exponential decrease in the number of states compared to normal deterministic finite automatons due to orthogonality and state clustering [1]. More accurately, statecharts have been shown to have exponential decrease in the number of states compared to alternating finite automatons (nondeterministic parallel automatons), which in turn are double exponentially more succinct than DFAs [1].

One interesting sidenote to exponential blowup problem is a recent result concerning the flattening of statecharts. The term flattening means removing hierarchy and retaining concurrency. This is different from expansion to a product machine [18] . It has been shown by Wasowski in [18], that for certain subsets of statecharts language, flattening is possible without explosions, i.e. the exponential blowup.

An obvious advantage of statecharts is their visuality, i.e. the ability to work on a more abstract level when defining a system. This property is made possible by hierarchy and orthogonality of states and general state transitions [4].

There are problems, with the statecharts and their semantics. Some of these were mentioned by Harel in [4]. These include semantical difficulty in the order of events generated by state transitions in the orthogonal components. Also cycles in are problematic. The extensions to handle time and temporal logic [14, 12], are also problematic and even erroneous in some cases. Some viable solutions have been proposed, however, by for example Leveson in [13] and Harel in [6]. Integrated development environment is also needed to take advantage of the hierarchical properties of statecharts and automated code generation.

6 Statecharts in STATEMATE

STATEMATE [9] is a set of tools for the development of complex reactive systems. Originally it was developed by Harel et al. and presented in [5]. Semantics of statecharts adopted in STATEMATE have been described in [6]. One notable fact is, that in spite of the semantical difficulties described in [13], the semantics of STATEMATE are sound and have been so from the beginning of the program development [6].

Graphical languages greatly increase the intuition and clarity in development of complex systems. [7, 5]. In STATEMATE development of reactive systems is divided to three different points of view: structural, functional and behavioral. These are represented by visual formalisms, i.e. graphical languages, one of which is the language of statecharts.

Stuctural view is used for a hierarchical decomposition of the system into modules. It also identifies the data that flows between different parts of the system. Structural view

Referred entity	Events	Conditions	Actions
State S	entered(S) exited(S)	in(S)	-
Activity A	started(A) stopped(A)	active(A)	start(A) stop(A)
Data items D, F	read(D)	$D = F$	D:=F
Condition C	true(C)		make.true(C)
Event E Action A $n \in \mathbb{N}$	timeout(E,n)		schedule(A,n)

Table 1. Some events, conditions and actions in STATEMATE

is described with the language of module-charts.

From the viewpoint of statecharts, *conceptual* view is more interesting. Conceptual view is further divided into two components: Functional and behavioral views. These views describe the activities and the flow of data and control activities of the system respectively.

Functional view is the functional decomposition of the system presented as *activity-charts*. One could say that activity charts are an extension to the DFDs, adding data stores and control activities. Data stores have an obvious meaning, as they represent the ability to store data items and produce them upon request. Control activities appear in the activity chart as empty boxes. Their contents, which constitute the behavioral view of the system, is modelled with the language of statecharts. Thus the behavioral and functional view are closely knit together.

In STATEMATE, control activities respond to events of the system. Several conditions for state transitions and the state of activities may be used in STATEMATE. Control activities may also start and stop activities as allowed in the statechart formalism. Also manipulation of data items is possible. Some available actions and activities are described in table 6.

The original idea about graphically clustering states into superstates from [4] has been implemented in STATEMATE. The user may select to view substates inside a superstate, thus *unclustering* its contents.

STATEMATE has support for numerous features, which make software development, verification and testing easier. Tests may be automatically generated for reachability, nondeterminism, deadlock and usage of transitions. Stepwise execution of model is also possible, and tester may define actions and events to take place during a certain timestep or change values of system variables or states. Model may also be translated to C -code. Formal verification is also possible [9].

There exists other tools, especially those for UML, which use the statechart formalism. These include, for example, Rational Rose [10] and Rhapsody [8]. Compared to STATEMATE, which was developed in the 1980's, these are relatively new and their syntax more UML inclined. On the other hand also STATEMATE has now support for UML diagrams.

7 Conclusions

The use of visual formalisms is increasing. Design methods, such as UML, from which automatic code generation is possible, are gaining more and more interest as products get more complex and their correctness must be formally verifiable. Proper formal proof of correctness enables the use of embedded software in safety critical products, such as life-support machines and nuclear reactors.

For complex reactive real time systems the language of statecharts is a strong design method. Statecharts and STATEMATE have been used successfully in military grade product definition and implementation [5]. Statecharts are also designed for development of real-time systems. Thus they do not suffer from the defects of design methods modified to support real-time development.

It is also good to remember that programmers, even the good ones, occasionally make mistakes. Usually this is measures as faults per lines of code (LOC). The amount of faults per LOC has a tendency to increase with the complexity of the product [17]. Thus the use of more abstraction, such as visual formalisms ,leads to increased productivity and correctness of products.

There still exists problems with visual methods of product development. Perhaps the most significant of these is ambiquity of semantics. Good syntax and semantics, paired with a development environment with clear and intuitive user interface, would establish a feasible alternative to traditional product development methods.

References

[1] D. Drusinsky and D. Harel. On the power of bounded concurrency i: finite automata. *J. ACM*, 41(3):517–539, 1994.

[2] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3), year 1997.

[3] H. Gomaa. Structuring criteria for real time system design. In *Proceedings of the 11th international conference on Software engineering*, pages 290–301. ACM Press, 1989.

[4] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[5] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. B. Trakhtenbrot. STATEMATE: A working environment for the development

of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[6] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[7] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pages 477–498, 1985.

[8] I-logix. *Rhapsody*. `http://www.ilogix.com/rhapsody/rhapsody.cfm`.

[9] I-logix. *Statemate*. `http://www.ilogix.com/statemate/statemate.cfm`.

[10] IBM. *Rational Rose*. `http://www-306.ibm.com/software/rational/`.

[11] J. C. Kelly. A comparison of four design methods for real-time systems. In *Proceedings of the 9th international conference on Software Engineering*, pages 238–252. IEEE Computer Society Press, 1987.

[12] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems 2nd International Symposium*, volume 571, pages 591–619, Nijmegen, The Netherlands, 1992. Springer-Verlag.

[13] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specifi cation for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

[14] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600, pages 447–484, Mook, The Netherlands, 3–7 June 1991. Springer-Verlag.

[15] Object Management Group. *Unifi ed Modeling Language (UML) version 1.5*. `http://www.uml.org/#UML1.5`.

[16] B. Sanden. Entity-life modeling and structured analysis in real-time software design - a comparison. *Commun. ACM*, 32(12):1458–1466, 1989.

[17] S. R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill Pub. Co., Fifth edition, 2002.

[18] A. Wasowski. Flattening statecharts without explosions. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 257–266. ACM Press, 2004.

Visual Programming Tools and Code Generation for Real-Time Systems

Samuel Siltanen
Helsinki University of Technology
Laboratory of Information Processing Science
saasilta@cc.hut.fi

Abstract

Real-time systems can be constructed using tools, such as I-logix' Statemate and Rhapsody, which utilize state-based visual formalisms. In this paper, their relationship to the development process is briefly covered. On the other hand, these tools can automatically generate code from the visual models. The diagrams that form those models are introduced from the code generation's point of view. The semantics of statecharts is explained more accurately, because they form the control part of the system. Finally, the techniques which have been utilized in generating the code from the models are discussed.

1. Introduction

Developing real-time systems can be very challenging. Often there are strict time constraints, computational resources are limited and reliability is an important issue at different levels of the system. Taking account these factors would suggest that the development tools should be able to provide a formal, yet intuitive way of designing the system. This would help the developers to understand the system more thoroughly and produce high quality software.

For humans, state-based models are a natural way of visualizing complex systems and thus understanding them, which has been noticed in various projects in which reactive systems were developed [2]. Thus Harel introduced a state-based modeling formalism called statecharts, which utilizes several innovative notations and a hierarchical approach to make the charts more compact and more intuitive than corresponding models presented as a state automaton. Statecharts have been utilized in the Statemate system, which is a development tool for complex reactive systems [3, 5]. This and other tools have made the use of statecharts quite common in real-time system design.

There is yet another benefit in statecharts: defining an accurate Semantics for them allows automatic code generation directly from the models. Code generation is possible using tools designed for that purpose such as I-logix' Rhapsody [6]. This forces the system specifications to be unambiguous, because otherwise the code cannot be generated without human intervention. In addition, the code can be directly mapped to the specification which embraces traceability. The code quality depends only on the generation tool, but not on programmers who are prone to make errors. The system can be simulated before the program code generation to assure correctness of the specifications or to test timing and possible anomalies. Even the development process itself is simplified, because it is supported by specialized tools and thus some parts of the process are automatic.

This paper first describes one tool chain which utilizes state-based modeling and can be used in the development of complex real-time systems. This includes a view of the development process and then a short description of the modeling concepts and their relationship to the automatic code generation. Then in the other section the code generation is discussed in detail. Solving the problems arising from the semantics of the statecharts are very essential to produce reliable and high quality code. On the other hand, there are several techniques to structure the generated program itself. After covering these aspects of the code generation, this paper draws some conclusions about the various approaches to the code generation and shortly ponders the problems still remaining.

2. Modeling with Statemate and Rhapsody

There are a variety of tools designed to help in the development of Embedded or real-time systems. In this paper, two of them are referred when discussing the tools for code generation, namely I-Logix' Statemate Magnum and Rhapsody in MicroC [5, 6]. Statemate system was described already in 1988 and was available two years later [3]. Thus it has been used in the industry for a long time and proved

to be useful in developing real-time systems. For generating of the production code, there is a separate tool, Rhapsody.

2.1. Tools and development process

Statemate Magnum and Rhapsody in MicroC together form a tool chain that covers the whole software development process from requirement analysis to coding and from unit testing back to the system testing. The software process is often described using the V-model (see Figure 1). Statemate Magnum is the tool used in requirements analysis and system level design and Rhapsody in MicroC is used in module level design and code generation. The models created by Statemate Magnum can be used in Rhapsody, and the test scenarios can be run using the code generated for the target system by Rhapsody. [6]

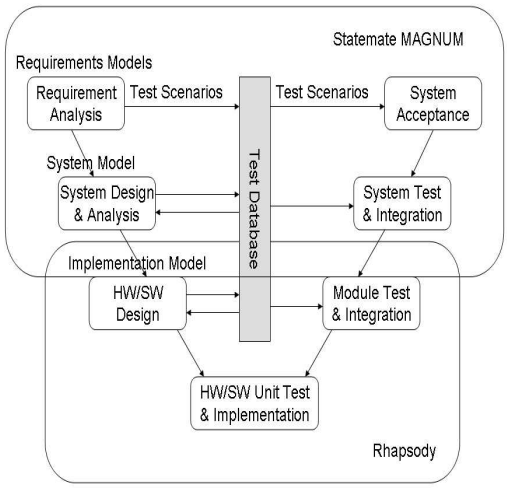


Figure 1. Statemate Magnum and Rhapsody form a tool chain which covers the whole software development process.

One of the most important tasks of Statemate Magnum is to provide an executable specification of the system. This is helpful when communicating with the customer to confirm that the specification meets their requirements. Thus the specification can be validated early in the development process which reduces the cost of corrections. The executable specification also helps the engineers to verify the behavior of the system. The specification can be graphically simulated and different scenarios can be run to see the interactions between the elements of the system. Those scenarios can be included in test plans and thus the test cases for the system testing can be directly mapped to the specification. In addition to these features, Statemate Magnum allows prototypes to be automatically generated from the models to the target language. Also the specification documents can be generated automatically. This can reduce the development time. [5]

Rhapsody offers a visual programming environment which is designed for real-time embedded system development. The emphasis is on software design and architecture more than on writing code at the lowest level. Thus Rhapsody uses several kinds of diagrams (i.e. statecharts) to describe the behavior of the real-time systems accurately. Those models can be analyzed and debugged at design level. The program code for the target operating system can be generated directly from them. For debugging purposes one can also add special interfaces to the generated code to monitor and control the code during the execution. The design can be thus easily validated. [6]

2.2. Modeling

Statemate Magnum and Rhapsody utilize several types of diagrams to model the system. Most important diagram types are activity charts and statecharts. There are also other diagrams which can be used to describe some details of the system or visualize it from another perspective. The purpose of all these diagrams is to model the system accurately to make automatic code generation possible.

Activity diagrams are used in modeling the architectural and functional properties of the system. They are hierarchical data-flow diagrams. The functional capabilities of the system are described as activities. The data elements and signals can flow between them. Activity charts are not dynamically committing, and thus they only model the possible activities and information flows, but do not tell when or why will something happen. On the other hand, they are useful in modeling the structure of the system. [3, 5]

Statecharts complement the activity diagrams by describing the behavior of the system. They can be thought to be the control part of the system. Thus the whole system can be modeled with statecharts that are assigned To activity charts (see Figure 2). One statechart controls the dynamics of the subactivities and the data-flow in an activity. This can include activating and deactivating activities, writing, modifying and reading data, sending signals and sensing when some of these things happen. In the code generation process the statecharts are the most important part of the model, because they describe the

exact behavior of the system while the other diagrams concentrate on the structure and different views of the system. [3, 5]

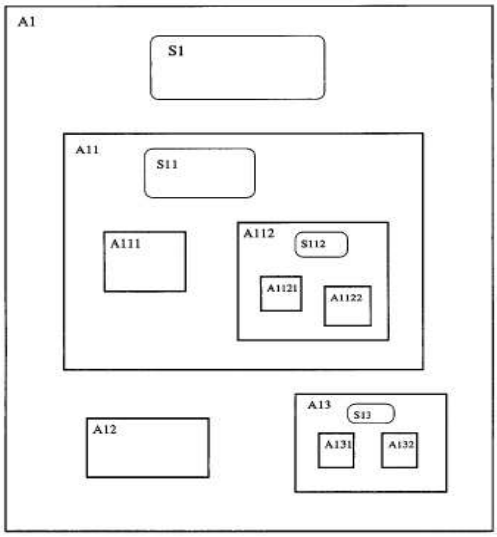


Figure 2. Activity charts (An) and statecharts (Sn) can be used to model the whole system.

There are also other diagram types. Flowcharts can be used to describe the algorithmic behavior of the real-time components and operations. Together with statecharts they can describe the behavior of the system efficiently. In Statemate Magnum also truth tables, time continuous or time discrete control diagrams and even C-code are allowed. [5]

3. Code generation

In this context, the code generation refers to the process where the visual formalisms are transformed into program code which can be executed on the target platform. Visually presented state-based models are often very intuitive for the designers of the system. This is not sufficient for generating the program code and thus the formal semantics of the model must be defined.

There are also some issues to be solved when the state-based model is converted into a model which can be represented in program code. Because embedded systems have often very limited resources, the program should not consume huge amounts of memory or make tedious calculation. In the real-time systems the time constraints may be strict, and thus the efficiency of the generated program is emphasized.

3.1. Statechart semantics

When statecharts were introduced by Harel [2], their syntax was defined in detail, but the semantics was left open. The original paper included only a brief discussion on the semantics, which mainly addressed some problems involved. Since then, there has been several proposals to the semantics of statecharts such as Pnueli's and Shalev's approach [8]. In his survey von der Beek compares some of them [9]. As stated by Harel there is no official semantics for statecharts, but researchers may choose the one which best fits for them [4]. However, the semantics used in Statemate is defined by Harel [4] unambiguously. This semantics is followed in this section.

Basically, statecharts consist of states and transitions between them. Those states can be basic states, which do not have substates or they can be compound states. Those compound states are whether AND-states, which include several orthogonal states, or OR-states, in which the substates exclude each other. Thus one must be in all substates of each AND-state and in exactly one substate of each OR-state. These states can form a hierarchical structure. Then the state of the whole system can be defined by enumerating the basic states which are currently active. This is called the basic configuration.

Each transition is triggered by an event. There can also be a condition, Which guards the transition from being taken unless the condition is true. When a transition is taken it can cause an action to be carried out. Because the system may have several active basic states, several transitions are usually taken simultaneously. The longest chain of these transitions that can be taken in a single step is called a basic compound transition. A full compound transition is a combination of basic compound transitions which lead to a full basic configuration (see Figure 3). Then the system is in a well-defined state and thus explaining the semantics of statecharts involves defining these transitions.

Because any transition can cause an action to be taken, it is important to define when that happens and how long does it affect. The execution order of the transitions may affect the resulting state. Thus Harel proposes that an action is taken in the step following the step during which the transition was triggered and it is active only during that step. This clarifies the concept of time in statecharts. In Statemate, there are two models of time: synchronous and asynchronous. In the synchronous model the system

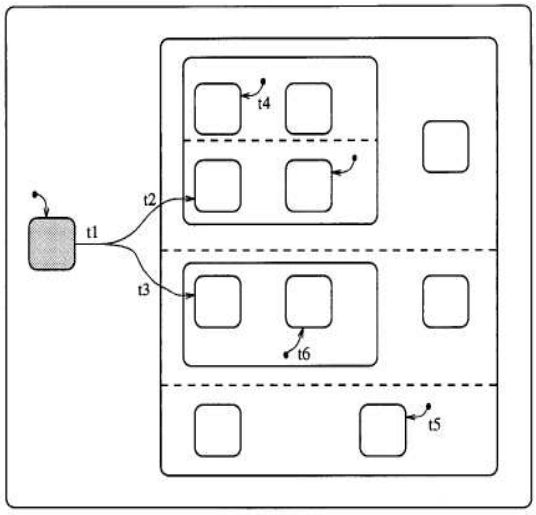


Figure 3. The initial compound transition [t1,t2,t3] must be Accompanied by the two continuation compound transitions t4 and t5 in order to form a full compound transition.

is expected to execute a single step every time unit and handle all external events that have occurred since the previous step. In the asynchronous model the system reacts to external events whenever they occur which allows several steps to be taken during one superstep. The steps themselves are assumed to take zero time. Statemate tools can generate code with both of the time models.

An interesting situation occurs when a value of an element is modified more than once or modified and read during one step. This is a racing condition and it must be solved to achieve unambiguous semantics for the statecharts. However, in a general case it is not clear how to do that and thus Statemate informs the user when a racing condition is detected, which allows him or her to clarify the problematic part of the model.

History connectors refer to the previous substate visited in a state. Their semantics is quite clear. If a state has history (e.g. it is not visited the first time or the history is not cleared), then if the connector is a regular history connector the most recently visited substate of the state is treated as the target of the transition which pointed to the history connector. If the connector is a deep history connector then the target of that transition is the most recent basic configuration of that state. If the state does not have history the target of the transition is the default state of the history connector if such state exists or otherwise the default state of the state in which the history connector is located.

When a transition is taken, it can pass through several levels of the statechart hierarchy. Thus it must be defined which non-basic states are exited and entered. The concept of scope introduced by Harel clarifies this issue. A scope is the lowest OR-state in the hierarchy that is a proper common ancestor of the sources and targets of the transition (see Figure 4). Then when the transition is taken all proper descendants of its scope before the transition are exited and all proper descendants of its scope after the transition are entered.

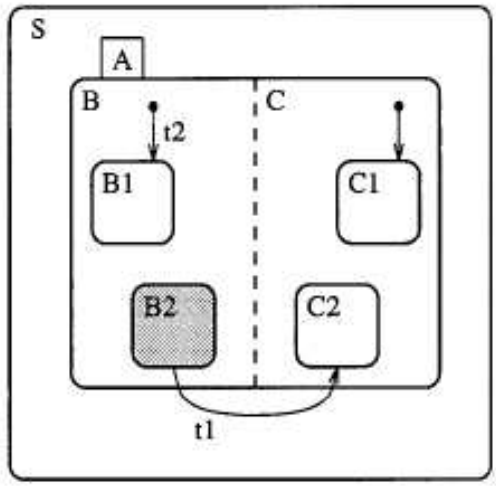


Figure 4. The scope of transition t1 is S which is an OR-state, but not A which is an AND-state.

There can be a conflict if there are two or more compound transitions that would leave a common state. Then if more than one of those transitions were triggered when the system is in that state, the behavior of the system is non-deterministic when choosing the transition if the conflict is not solved. Often the conflict can be solved by using scopes. Priority is given to the transition whose scope is higher in the hierarchy. If the scopes are equal, then in the code generation tool selects the first possibility it finds and the user can be informed that there is non-determinism in the system.

Using this semantics leads to the following algorithm, which is used in The simulation tools in Statemate, and

which thus defines the behavior of the generated code:

The Inputs:

1. The status of system including
   - a list of states in which the system currently resides
   - a list of activities that are currently active
   - current values of the conditions and data-items
   - a list of events that were generated internally in the previous step
   - a list of scheduled actions and their time for execution
   - a list if timeout events and their time for execution
   - relevant information of the history of states
2. The current time
3. A list of external changes presented by the environment since the last step

The output:

1. A new system status

The algorithm:

1. Step Preparation
   - Add the external events to the list of internally generated events
   - Execute all the actions implied by the external changes, but do not take any transitions
   - Test if there are scheduled actions whose time is less or equal to the current time and execute such actions and remove them from the list of scheduled events
   - Go through timeout events and check if their condition event is generated; if it is, set their timeout time to the current time added to the timeout delay; else if the timeout time is less or equal to the current time generate the timeout event and set the timeout time to infinity
2. Compute the Contents of the Step
   - Compute the set of enabled compound transitions
   - Remove from this set all the compound transitions that are in conflict with an enabled compound transition of higher priority
   - Split the set of enabled compound transitions into maximal nonconflicting sets (*)
   - For each set, compute the set of enabled static reactions defined in states that are currently active and are not being exited by any if the compound transitions in the set
   - If there are no enabled compound transitions or static reactions the step is empty, else test if the stage (*) resulted a single set which would then constitute the step; otherwise the set selection is non-deterministic
3. Execute compound transitions and static reactions
   - For each static reaction in the set of enabled compound transitions, execute the actions associated with it
   - For each compound transition in the set of enabled compound transitions
     - update the history of all the parents of the exited states
     - delete the exited states from the list of states in which the system resides
     - execute the actions associated with exiting the exited states
     - execute the actions associated with the compound transitions
     - execute the actions associated with entering the entered states
     - add the entered states to the list of states in which the system resides

3.2. Structuring the generated code

There are several methods for transforming the statecharts into program code. Because in statechart models the states are in hierarchy, several states can be active simultaneously and the transitions between them might have complex dependencies [2], the model must be simplified to obtain a model which can be represented in conventional programming languages.

Both Wasowski [10] and Pinter [7] suggest a model which preserves the hierarchy in the model by explicitly storing the information about active non-basic states. The benefits of this approach are that the program structure is relatively intuitive which makes it easier for humans to maintain or develop further the computer-generated code. Also the program size grows only linearly when the number of states and transitions grows in the statechart. One disadvantage is that the generated program is inefficient in smaller models, because of additional work is done to handle the hierarchy.

Statecharts can also be transformed into a finite state automata which does not contain any hierarchy. The problem in this approach is that the number of states grows exponentially when the number of states and transitions grows in the statechart. Wasowski tries to tackle this problem in his latest work [11]. The main benefit of state automata is that they are very simple and can implemented very efficiently on almost any computing platform.

In the discussion of the common implementations of the statechart code generators Pinter [7] also mentions other techniques such as nested switch statements, action-state tables, usage of state design pattern, and quantum hierarchical state machine. The idea of nested switch statements is to have an event handler with a switch statement with a branch for each state. Then in each branch there is another switch statement with branches for each possible event which include the actions to be taken. Disadvantages of this technique are that it produces code that is hard to read and the entry and exit actions of states are difficult to implement correctly. On the other hand, the technique is quite straight-forward, and even the code generated by I-Logix' Rhapsody is structured using a similar approach.

Action-state tables are a technique were there is a table which is indexed by both the current state and the event that has occurred, while the table entries are pointers to functions similar to the inner branches in nested switch statements. This approach is faster but requires more memory than the nested switch statements. The generated code is still obscure and many features of the statecharts (i.e. history, concurrency) cannot be implemented.

An object-oriented approach to generating code from statecharts uses the state design pattern [1]. There is a common interface class for states which has an event handler. The inherited states implement that function and thus the code of each state is encapsulated in a separate state class. This technique produces structurally better code than the previously mentioned techniques, but it still cannot represent the dynamic properties of statecharts due to the static nature of the class structure.

Quantum hierarchical state machine is a pattern where there is a base class which implements an event dispatcher and a transition function. This class is inherited by classes which handle the events in specific states. The state hierarchy is preserved, because the events are handled in the basic states and delegated to their parent states, which are explicitly stored, if necessary. This allows implementation of the entry and exit actions in transitions. Also the history information can be saved in the parent classes. Concurrent operations can be supported by decomposing and-states into individual state machines. Pinter [7] suggest several optimizations to make this approach Feasible for generating program code for systems with limited resources.

4. Conclusions

Using tools that are designed for modeling, simulating and debugging software for real-time systems may speed up the development process. They can be used to generate the program code automatically to avoid unnecessary work and human errors. On the other hand, the readability of the generated code is still a problem. This is mainly due to the structuring of the code which simulates the state-based model. To make the code more readable more abstraction levels should be used instead of switch-case-statements. It is not clear how this could be done automatically. If the whole system could be programmed and maintained using the design tools, the readability would not be an issue. Then the emphasis would be on higher level design and modeling the system using visual formalisms such as statecharts. Obviously, this is the tendency in the future.

Statecharts have proved to be effective in modeling complex real-time systems. Together with activity charts they are expressive enough to define the structure and behavior of the whole system. Thus only algorithmic details and machine specific features require other diagrams or program code segments. The semantics of statecharts have been defined accurately and there are only some special cases in which the definition allows ambiguities. Such cases are usually indications of design errors and thus they are reported by the tools. If the semantics of statecharts as specified by Harel [4] are compared to that of the traditional programming languages (i.e. C), one could even state that statecharts are a more accurate way of defining programs. Also because statecharts are very intuitive and can be used at higher levels of design, design errors are less likely than when using separate modeling and programming languages. Thus statecharts can provide the reliability required in real-time systems.

Timing issues are a very integral part of statechart formalism. The algorithm presented when discussing the semantics of statecharts did most of its work in determining occurrence of the scheduled and timeout events, in addition to handling external events and finding the full compound transitions. Also two timing models were supported which allow two types of behavior. First, in the synchronous model the system was guaranteed to always be in a stabile state. Then, in the asynchronous model the system reacted faster to the external events, but it was possible that the system was not always stabile. However, because in

Statemate the user can choose which model to use, the code generated from statecharts can usually handle the timing sufficiently well in most applications. The wide range of real-time operating systems supported by

Rhapsody suggests that this approach is feasible in developing real-time systems.

It is obvious that the statechart formalism will remain as one of the Most powerful design techniques for real-time systems. Also the Statemate-Rhapsody tool chain has been used in several successful projects where complex real-time systems were developed. Because there are not many competing tools to seamlessly cover the whole development process, this tools chain will be the best one available for many real-time system projects.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):213–274, 1987.

[3] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[4] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), 1996.

[5] I-Logix. Statemate magnum. `http://www.tess-com.it/brochures/SW/StatemateBrochure(V3).pdf`, 2000.

[6] I-Logix. Rhapsody in microc. `http://www.tess-com.it/brochures/SW/RhapsodyinmicroC.pdf`, 2001.

[7] G. Pinter and I. Majzik. Program code generation on uml statechart models. *Periodica Polytechnica*, 47(3-4):187–204, 2003.

[8] D. Pnueli and M. Shalev. What is a step: On the semantics of statecharts. In *Proceedings of the Symposium on Theretical Aspects of Computer Software*, volume 526, pages 244–264. Lecture Notes in Computer Science, 1991.

[9] M. von der Beek. A comparasion of statechart variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863, pages 128–148. Lecture Notes in Computer Science, 1994.

[10] A. Wasowski. On effi cient program synthesis from statecharts. In *Proceedings of the 2003 ACM SIGPLAN/SIGBED conference on Language, compiler and tool for embedded systems*, number 7 in 38. ACM SIGPLAN Notices, 2003.

[11] A. Wasowski. Flattening statecharts without explosion. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers and tools*, number 7 in 39. ACM SIGPLAN Notices, 2004.

On Formal Modeling of Hybrid Systems

Antti Kantee
Helsinki University of Technology
Laboratory of Information Processing Science
antti.kantee@cs.hut.fi

Abstract

Despite the discrete minds of most computer scientists, computers and computer programs ultimately operate in a real, continuous world. Most "normal" programs which take input, do a calculation, and produce an output can completely be modeled by using a discrete model. However, this is not the case for reactive systems. In reactive systems it is commonplace to operate on a continuous signal, which is adjusted according to the programmer's desire. The produced signal is fed as input and again adjusted to meet the requirements. It is possible to quantasize a continuous signal into discrete steps with some loss of accuracy, but it is not possible to remain accurate, as this would require an uncountably infinite number of states in the discrete model.

This paper deals with modeling paradigms applicable to systems with both a continuous and discrete variables, known as hybrid systems. The first part involves a theoretical model, while the second part takes a more "hands-on" approach presenting both code generation from a model and existing tools for the job.

1. Introduction

Control theory is a well-understood form of engineering. It involves calculating the controls u required to satisfy the system dynamics. For dynamics $x' = f(x, u)$, where x is the system current state, control theory is interested in creating a control law $u = g(x)$. To calculate this law in a discrete world of computer systems, a sampling period must be decided on to define the points when x is sampled and the control u is calculated. This dictates a loss of accuracy from the ideal situation to the real world.

However, even though our real world implementation must by necessity be an approximation of the system itself, this is by no means true of the model used to describe the system. By constructing the modeling framework so that it is able to deal with continuous variables, we allow the model to remain an accurate depiction of the real world, and lose pure mathematical accuracy only at a later stage in the process.

If the system with continuous variables also features discrete variables, we are dealing with a hybrid system. In such a system the discrete signals are usually used to control the differential or algebraic equations imposed onto the continuous variables.

The applications of hybrid models can be found in all places where computers are used to control analog signals. Especially useful formal models are in areas of industry where rigorous and robust implementations are necessitated by the hard real-time characteristics of the operating environments. There formal models can be used to either make the design undergo vast tests using simulation, or even generate the target code from the model. Classical users are the automobile, aviation and space flight industries.

The structure of this paper is as follows. Chapter 2 consists of an introduction to the formal modeling paradigms. It starts out with simple transition systems, and finishes off with a phase transition system capable of expressing a hybrid system. Tools available for modeling hybrid systems in addition to the requirements and implications of generating code from the models are discussed in Chapter 3. The paper concludes in Chapter 4.

2. Hybrid Modeling

First, let us analyze continuous variables from a mathematical perspective. Between any two given points in time, t_0 and t_1, where $t_0 \neq t_1$, a continuous analog variable has an uncountably infinite number of values. In contrast, a discrete signal will only have a finite number of values between the same two given points in time. The discrete case can be dealt with a state-based model, where, in the extreme case, every possible value is represented by a state in the model. For the continuous case, this is obviously not an option, and more powerful formalisms must be developed.

2.1. Traces

Let us start considering modeling by simply considering the computation that a system produces. A computation or trace σ of a system can be formally represented by a series states the system goes through. A state is of course nothing more than a binding for the system's variables, and therefore I denote it with v.

$$\sigma : v_0, v_1, v_2, ...$$

We can call a trace an abstract model of a program [3]. While it does not give us any details on how the program is internally constructed, it does provide us with a detailed description of what the program does.

Since we are operating in the domain of real-time programming, simply having the states the program visited is not enough. We also require the information of when a state was visited, since that is part of the operating specification. To address this problem, the system states are augmented with time information to produce *situations* [9], which we can record in a timed trace:

$$\sigma_t : s_0, s_1, s_2, ...$$

This series of situations is required to be a progressive sequence: for every $i \geq 0$, $t_{i+1} \geq t_i$. This of course is a natural ordering for the sequence, since time usually progresses in the above manner. Notice that the next time value is not required to be strictly greater. We will get back to this later.

Now we are beset by the lack of formal operators in our basic repertoire. We require new logical operators and constrains to be able to check verify that the program agrees with the specification.

2.2. Temporal Logic

The use of temporal logic in the specification of programs was first suggested by Amir Pnueli [13] and has been since extended [9]. The operators of temporal logic are presented next. More in-depth descriptions can be found in appropriate literature [10].

For a given formula φ the following operators are defined:

$$\Box\varphi$$

This is the safety properly. It states that φ shall hold for all states after the current state.

$$\Diamond\varphi$$

This is the liveness property. It states that φ shall be true at some point later in the sequence.

It is possible to form various combinations of the basic operators. For example, it is important to realize the difference between $\Diamond\Box$ and $\Box\Diamond$, which respectively translate to "will eventually always hold" and "will always hold at some points in time", in intuitive terms. Using a negation is also possible, for example $\Box\neg\Diamond\varphi$ means that there will always exist a future point in the trace where φ does not hold, i.e. φ will not hold infinitely many times.

It may also be necessary to access individual terms in formulas. For a given term r, the denotations $\bigcirc r$ and $\ominus r$ denote the next and previous values for r, respectively. The age of a formula in turn is the time value indicating for how the long the formula has continuously been true. The notation used is $\Gamma(\varphi)$. As a special case, if φ does not hold, $\Gamma(\varphi)$ is defined to be 0. These two cases are among the extensions introduced by Maler, Manna and Pnueli [9].

After mastering only the above, we are definitely still neophytes in the art of temporal logic. More complex operators, for example interval-bounded operators such as $\Diamond_{(a,b]}\varphi$ exist. However, getting into detail on them is beyond our scope. The interested reader is directed to literature on the subject [3, 9, 10, 13].

2.3. Hybrid Models

As was already established, a hybrid model describes a system with both discrete and continuous components. Such systems are not essentially more complex than state transition systems, but to ease the learning curve, I will start with a short introduction on three concepts probably familiar to the average reader, and augment them with the concept of time.

Formally the representation of models is fairly dense: a model is simply an n-tuple consisting of various sets: $M = (P, Q, A, c, T)$[1]. By tying the symbols in the n-tuple to stand for states, entry and exit points to them, system input and output variables, transitions and the conditions under which they can be taken, and by adding some rules on how to use these variables we will have a created a system capable of formally describing hybrid models.

2.3.1 Timed Statecharts

Timed statecharts is an extension to Harel's original specification of statecharts [6]. Familiarity with statecharts in general is recommended, since we will later in the paper discuss concepts which have been heavily influenced by them. In case the original specification [6] is too verbose, introductory-level summaries [5] also exist.

A timed statechart [7] is simply a statechart with a time interval (l, u) associated with each transition. The lower

[1] In this example the letters in the formal presentation do not stand for anything.

bound signifies the minimum time that must be spent in the current state before a transition can be taken, while the possibly infinite upper bound limits the time during which the transition must be taken, if it is to be taken at all. It is easy to see how these constraints relate to deadlines in real-time systems.

2.3.2 Timed Automata

A timed automaton in turn is a finite state automaton with clock information included. It functions similarly to a "normal" case, but in addition to using "normal" input, it can make state transitions also based on the state of the clock or clocks present in the system. The timed automaton will accept or reject input based on these two input sets. As is the case with vanilla finite state automata, timed automata can be deterministic or non-deterministic.

While not the original source, a good and more formal description of timed automata can be found in the real-time modeling survey by Alur and Henzinger [3].

2.3.3 Timed Transition Systems

Since the concept of hybrid systems will be built upon the foundation of timed transition systems, I will go over them in more formal detail than what was given for timed statecharts and timed automata. The notations used here are similar to the ones used by Maler, Manna and Pnueli [9].

A timed transition system is formally a 5-tuple $S = (V, \Theta, \mathcal{T}, l, u)$, which essentially is a normal transition system $S_0 = (V, \Theta, \mathcal{T})$, to which terms indicating lower and upper bounds of transitions (l and u, respectively) have been added. An overview of the tuple members is given next:

- V: the finite set of *state variables*. Every member of V can be assigned a value. All the possible combinations of values assigned to the members V is denoted by Σ. Additionally, Σ_t augments the assignation with a time value to produce situations.
- Θ: the possible initial states of the system.
- $\mathcal{T}$: a finite set of transitions. A transition is simply a function from the interpretations of V to all possible targets for the transition. Mathematically speaking, this can be expressed as [2]: $\forall \tau \in \mathcal{T}, \tau : \Sigma \rightarrow 2^{\Sigma}$. The targets of the transitions for the current state s are called the τ-successors of the state s. Additionally, a transition is said to be *enabled* iff $\tau(s) \neq \emptyset$, otherwise it is said for be *disabled*.
- l: the lower time-bound for a transition. if $\tau \in \mathcal{T}$, then l_τ signifies the minimum time that the transition must be enabled before it can be taken
- u: the upper time-bound for a transition. This signals that a transition $\tau \in \mathcal{T}$ does not stay enabled longer than u_τ. It is required that $u_\tau \geq l_\tau$, but an infinite upper bound is allowed.

A timed transition system is capable of producing a computation. As already was presented earlier, the computation produced by the system is given by a timed trace:

$$\sigma : s_0, s_1, s_2, ...$$

The key for the system to get from stage σ_i to σ_{i+1} is of course $\tau \in \mathcal{T}$. It is important to notice that the mapping between the timed transition system T and the produced computation σ is not a bijection. For example if we have a timed transition with a single transition having no other limitations besides the time limits $(1, 2)$ for the transition, we already have an uncountably infinite number of possible computations resulting from the single timed transition system.

2.3.4 Hybrid Systems

A framework for the formal specification of hybrid systems was proposed by Maler, Manna and Pnueli [9]. Conceptually the system is very similar to the timed transition system described in the previous section.

First of all, to accommodate for the presence of both continuous and discrete variables, the notion of time should be reinvestigated. Since dealing with a discrete variable is theoretically instantaneous, it will take up no time in the model[3]. And on the contrary, since dealing with a continuous variable is happening "all the time", time should be advancing continuously in the model, not in discrete steps.

Time is therefore divided into two different *moments*, called *discrete moments* and *continuous moments*. Discrete moments do not advance time at all.

Since time does not advance during a discrete moment, we cannot use the otherwise manifest real number ordering relation on time to impose an ordering relation for the trace: the order of the trace during discrete moments would not be defined in that case. Here, ordering is defined by a *time structure* induced by the time sequence $\theta : t_0, t_1, t_2, ...$:

$$T_\theta : \{< 0, t_0 >, < 1, t_1 >, < 2, t_2 >, ..\}$$

[2] The fact that the image of τ is the power set of Σ, denotated by 2^{Σ} or $\mathcal{P}(\Sigma)$, instead of plain Σ may seem strange at first, because the transition simply takes the system from one interpretation to another. However, the purpose of the power set is to emphasize the fact that the transition maps the system to a state which contains some limited selection, not necessarily all of the interpretations from Σ.

[3] Of course it will take up some time in the actual computation, but it can (hopefully) be considered to be negligible. If discrete variable computation starts taking up significant time, there is probably something wrong with the design. After all, discrete calculation in a hybrid model is mostly overhead, and the real work is done by continuous-type computations.

Especially notice that nothing dictates that $t_0 \neq t_1 \neq t_2$, and the values would in fact be equal if the system was in a discrete moment. An ordering on the set is defined by:

$$< i,t > \prec < i',t' > \iff i < i' \bigvee t < t'$$

Notice that the sequence θ is required to be progressive (i.e. non-decreasing), and therefore T_θ which contains the pairs $< i,t >$ and $< i',t' >$, where $i <' i$ and $t > t'$ is impossible.

Now we can define a *hybrid trace* as a pair of sequences (θ, σ), where θ is the progressive time sequence and σ assigns the system a state in every (discrete and continuous) moment in the time structure T_θ. I will discuss and extended this a bit further in the next chapter on phase transition systems.

Variable updating follows a simple rule: discrete variables remain constant during continuous moments, and continuous variables remain constant during discrete moments.

2.3.5 Phase Transition Systems

In hybrid systems we have the ability to incorporate both discrete and continuous variables into the model. However, the model in the previous chapter lacks one fairly important detail: it does not specify how continuous variables are adjusted. To take care of this, Maler, Manna and Pnueli introduced Phase Transition Systems [9].

A phase transition system Φ is a tuple $(V, \Theta, \mathcal{T}, \mathcal{A}, l, u)$. In more detail, the system is composed of:

- V: the system variables. Here the set consists of both discrete and continuous variables: $V = V_d \cup V_c$.
- Θ: set of initial states.
- $\mathcal{T}$: the set of transitions, similar to ones in timed transition systems. It should be noted, that a transition is allowed change the values of discrete and continuous variables alike. This makes sense if we for example think of a situation where we want to set the initial signal value to 0 when the system makes a transition from one mode to another.
- $\mathcal{A}$: a set of *activities* used for controlling continuous variables. These are usually of the form $a_\alpha \rightarrow \mathcal{E}_\alpha$, where a_α is a discrete boolean activation condition for the differential equation given by $\mathcal{E}_\alpha$.
- l: the lower bounds for transitions.
- t: the upper bounds for transitions.

Now our computation consists of a trace (θ, σ), where θ is the progressive time sequence and σ is a mapping binding the state of both discrete and continuous variables. As was established, during continuous moments discrete variables remain constant. The values of continuous variables during continuous moments are of course not enumerated, but rather their values are given by the associated activities. For discrete moments, the mapping σ works as before.

3. Tools and Code Generation

The advantages and problems of automatic code generation from hybrid models were well summarized by Maler [8]. I will present some of his ideas here before moving on to actual examples dealing with code generation.

It is important to keep in mind that there are two different target platforms for code generation: the simulation platform and the actual target platform. These two may have wildly different hardware characteristics.

3.1. Motivation for Automatic Code Generation

Creating an accurate model describing a system takes some effort. If the modeling is done using a system with formal and exact semantics with a clear mapping to a programming environment, there is no theoretical obstacle in creating code from the model.

The advantages of automatically creating code are several. First of all, the produced code will always exactly follow the model, instead of being a programmer's (possibly misguided) interpretation of it. In addition to interpretation errors, automatic code generation will rule out programming errors. Although this defeats the purpose of unit testing, testing for logical errors in the model is still required. Second, code generation does not take any human effort. Third, if the specification is changed, it is easy to generate new code from the updated specification.

The disadvantage of automatic code generation is similar to problems with all "smart" tools. Instead of using a tool for what needs to be done, it is possible for the user to end up in a situation where she needs to trick the tool into doing her bidding. This kind of trickery of course relies on accidental behavior of the tool, and may break in later versions. Also, since the tool operates on a set of strict formal parameters, it may not be capable of the same level of optimizing that a human is.

3.2. Simulation vs. Code Generation

I already mentioned earlier that code generation for a simulator and the actual target platform are two completely different issues. Code generation for a simulator allows for a much more "sloppy" translation. The simulation usually runs on large workstation hardware with possibly hundreds

of times the hardware resources when compared to the target system. Also, time does not really exist when running the simulation: it is an illusion created by the simulation.

Therefore, code creation from a model to a simulator platform is a different issue from code creation to the target platform. The latter includes timing and hardware constraints which do not concern the former.

This is of course not to say that code generation for the simulator alone is not an interesting or useful task. The simulation can be used to verify the principles of operation of the software and do rapid prototyping. The actual production code can be written by hand once the principles of operation have been verified in the simulation.

3.3. CHARON

The CHARON language [2] was developed at the University of Pennsylvania for the modular specification of hybrid systems. On a basic level CHARON is similar to what we saw earlier: a system making transitions from one state another, and when no state transition is active, time advances.

CHARON is made up of building blocks called *agents*. Agents can exist either as atomic agents or composite agents made up of atomic or other composite agents. Agents can communicate with each other by using a set of shared variables. Mathematically we can denotate a composite agent as a non-empty set of sub-agents and as a set of variables $A_c = (SA, V)$. An atomic agent in turn consists of a *mode* and variables: $A_a = (M, V)$.

The behavior of agents is controlled by *modes*. A mode is also a hierarchial entity, which can consist of submodes. The formal representation of a mode is as follows: $M = (E, X, V, SM, Cons, T)$. In more detail, the tuple members are sets of:

- E: entry control points
- X: exit control points
- V: variables
- SM: submodes
- $Cons$: constraints. These control how continuous variables are adjusted when time passes. They are much like the *activities* of phase transition systems.
- T: transitions, containing both a guard and an action

It is not difficult to see how the hierarchial nature of modes in CHARON closely resembles the idea of statecharts [6]. To understand the operation of modes in CHARON, it is probably wisest and easiest to start by studying the operation of statecharts.

Having both agents and modes as separate entities in the model might seem like a confusing concept, since they both are hierarchial objects that contain variables. However, by encapsulating modes inside agents, it is possible for modes to be complete hierarchial: parallel composition is handled by having multiple parallel modes inside an agent at the appropriate level in the agent-hierarchy.

The variables in CHARON are divided into two distinct sets: discrete variables and continuous variables. The fact that variables belong to a certain agent may be emphasized by using the notation $A.V$ for the variable set V related to agent A. The variable set is divided into the sets of discrete variables ($A.dscV$) and analog variables ($A.anaV$). In CHARON the set $A.anaV$ is subject to a further subdivision into algebraic variables ($A.algV$) and differential variables ($A.diffV$). Differential variables take part in continuous differential equations, while algebraic variables take part in equations without a differential twist. The two sets are required to be disjoint. To enumerate all of the above:

- $A.V = A.anaV \cup A.dscV$
- $A.anaV = A.diffV \cup A.algV$
- $A.diffV \cap A.algV = \emptyset$

Variables can also be divided into local variables and global variables, and readable and writable variables. However, the definitions are intuitively clear, and going over their semantics more accurately would not serve our topic of hybrid systems in general.

Variable scoping is another topic related to the hierarchial model of CHARON. For a programmer hierarchial scoping is immediately clear: global variables may be accessed by anyone, while local variables are accessible only to the mode and its submodes.

The passage of time is handled similarly to what we saw earlier in phase transition systems. During an *update round* the agent undergoes transitions from one mode to another. Once all the discrete transitions have been taken and none are enabled, time is allowed to pass.

3.3.1 CHARON and Code Generation

Now that we have seen what CHARON looks like, and noticed some similarities with the hard theory on the subject, it is time to look at generating code from a CHARON model [4].

Code generation in CHARON takes place in two steps. First, a C++ program is generated from the model. After that, it is a simple matter of compiling the C++ code to the target platform to obtain working machine code. CHARON concepts clearly map to classes and inheritance in C++, but since those can be ”simulated” in plain C also, one cannot but wonder would plain C have been a better choice.

The problem in generating code is of course the necessary quantization of continuous variables: computers and therefore computer programs operate in a discrete universe, there is simply no way around it. This fact also makes transforming the discrete variables from the model to code a fairly non-interesting feat, and I will simply skip it.

When abandoning mathematical modeling wonderland, the best we can do is to minimize the inaccuracies resulting from quantization. By using a finer granularity in sampling the signal more accuracy is gained, but processing time is lost. The key is to select the sampling period so that the system does not miss any state transformations.

The errors in quantization inaccuracies introduced when moving from the model to the actual world have been analyzed in-depth using a mathematical model in the original paper [4]. Since the mathematical model requires introducing numerous new terms, I will not go through it. The interested reader is directed to the source.

Code generation must also take care to retain correct evaluation order. If variables depend on other variables (possibly) in other agents, the values of those variables must be evaluated first.

The specific actions necessary for transforming variables, constraints, invariants, transitions, control points, modes and agents into code are presented by Alur et al [4], and the interested reader is encouraged to look there for details.

3.4. Simulink / Stateflow

The Simulink package sold by MathWorks is a professional toolkit for simulation and code generation. It is used by companies such as Boeing, DaimlerChrysler and the U.S. Air Force [1]. The purpose of this chapter is to give the reader an idea of what can be accomplished with Simulink and Stateflow. The discussion will take place on a very general level; further details can be extracted from respective user guides [11, 12] or from material discussing the operation of the tools under the hood [14].

Simulink lends itself to creating a hierarchial model of the target system, which can be used for simple modeling, simulation or analysis. It is not difficult to notice the huge influence of statecharts [6] also here. This influence is is even more eminent in Stateflow blocks.

Modeling in Simulink is done through a graphical user interface, which promises to be a practical interface for building hierarchial models. Most of the work is done with a mouse by clicking around the model, creating various pieces of the model, and dragging them into place. Viewing the constructed models at any level of the hierarchy is possible, so getting both a detailed and general image of the system is possible.

In addition to a library of ”standard” construction blocks, it is also possible to define custom blocks. If it is not possible to build the blocks out of more primitive blocks available, it is possible to give a written programmatic description of the operation of the block. The programmatic approaches are called *S-functions*, and can be written either in MATLAB or C. Another way to view S-functions is as an enabler giving the possibility to add already written C or MATLAB code to the simulation, i.e. simulate running the code and see how it interacts with the environment.

To add discrete state-type modeling capabilities to Simulink, Stateflow can be used. If we consider the division we made to variables in Chapter 2, Stateflow would be concerned with V_d. Similar to the master Simulink package, Stateflow is controlled with a graphical user interfaces, and models are hierarchial.

For Stateflow to be of any use, it of course must be possible to integrate the Stateflow model into the Simulink model, and influence the interaction between the two. Stateflow blocks always operate in an event-driven manner. These events can be either internal to the Stateflow block, or they can be propagated from the Simulink model. In addition to be able to define input events into Stateflow blocks, it is of course equally important to be able to define output events, so that it is possible to change the operation of the Simulink model based on a control event that was processed within the Stateflow block.

The final interesting task for the toolset is code generation from the model. The Realtime Workshop Embedded Coder and Stateflow Coder can be used to generate C code from Stateflow blocks and discrete-time Simulink blocks for a given set of target environments.

4. Conclusions

This paper introduced the the modeling paradigms and tools available for use with hybrid systems. It was discovered that to accommodate for the nature of analog variables, variables need to be partitioned into discrete and continuous variables, $V = V_d \cup V_u$, which receive separate treatment. The passage of time in the models had to be adjusted to suit this: time is partitioned into discrete and continuous intervals. Both sets of variables can be adjusted during discrete intervals, but time does not advance. During a continuous interval time advances and continuous variables are adjusted according to the equations related to them.

The tools used to model hybrid systems draw heavily not only from the theory of hybrid models, but also from the hierarchial concept introduced by statecharts. Tools are available in both professional and academic capacities. In addition to creating an accurate simulation out of the hybrid model, which can be used for rapid prototyping and general verification of the design, some tools can be used to generate code for the final target platform.

References

[1] Mathworks simulink 6.1 user stories. *http://www.mathworks.com/products/simulink/userstories.html*. Referenced 29.11.2004.

[2] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specifi cation of hybrid systems in CHARON. In *HSCC*, pages 6–19, 2000.

[3] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, volume 600, pages 74–106, Mook, The Netherlands, 3–7 June 1991. Springer-Verlag.

[4] R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky. Generating embedded software from hierarchical hybrid models. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 171–182. ACM Press, 2003.

[5] M. Byckling. Introduction to statecharts in design of reactive systems. 2004.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[7] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems 2nd International Symposium*, volume 571, pages 591–, Nijmegen, The Netherlands, 1992. Springer-Verlag.

[8] O. Maler. Code generation for on-board satellite software.

[9] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600, pages 447–484, Mook, The Netherlands, 3–7 June 1991. Springer-Verlag.

[10] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specifi cation*. Springer-Verlag, 1992.

[11] Mathworks. *Stateflow User's Guide*.

[12] Mathworks. *Using Simulink*.

[13] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, Oct. 31–Nov. 2 1977. IEEE, IEEE Computer Society Press.

[14] A. Tiwari. Formal semantics and analysis methods for simulink statefbw models.