UNIVERSITÄT SALZBURG

A Java Framework for Giotto

Markus Amersdorfer Helge Hagenauer Werner Pohlmann

Technical Report 2005-01 April 2005

Department of Scientific Computing

Scientific Computing

Jakob-Haringer-Straße 2
5020 Salzburg
Austria
www.scicomp.sbg.ac.at

Technical Report Series

A Java Framework for Giotto

Markus Amersdorfer
FB Scientific Computing
Universität Salzburg
Jakob-Haringer-Straße 2
A-5020 Salzburg, Austria
mamers@cosy.sbg.ac.at

Helge Hagenauer
FB Scientific Computing
Universität Salzburg
Jakob-Haringer-Straße 2
A-5020 Salzburg, Austria
hagenau@cosy.sbg.ac.at

Werner Pohlmann
FB Scientific Computing
Universität Salzburg
Jakob-Haringer-Straße 2
A-5020 Salzburg, Austria
pohlmann@cosy.sbg.ac.at

ABSTRACT

Real-time programming in Java (RTSJ) relies on threads for concurrency, which are difficult to use and generally suffer from nondeterministic execution. We propose to define and implement frameworks that offer programming models which are simpler in use though possibly more restricted in scope. As an example, we describe a framework that supports the Giotto programming model in Java. Giotto ([13]) is a new language for embedded software; it is time-triggered, with temporally fixed communication events and computations occurring in between.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

Keywords

Embedded systems, real-time systems, time triggered systems

1. INTRODUCTION

Some years ago, using Java with its object-oriented features, automatic garbage collection etc. for real-time programming may have seemed a strange idea, but with the Real Time Specification for Java (RTSJ, [3]), things and attitudes have changed. RTSJ is now seen as a serious and in some respects superior rival to Ada ([4],[20]). This comparison is well founded: both languages rely on threads (or tasks) for concurrency.

Concurrency is characteristic of real-time software, which has to deal with several things at once. And threads, together with coordination techniques like monitors, are an old and extensively studied concept that seems a natural extension of imperative programming. On the other hand, threads are notorious for hard-to-understand programs and hard-to-find bugs which sometimes come up only after years of happy use. The basic semantic idea – threaded software gives an arbitrary interleaving of the respective sequences of actions – exhibits the risk of race conditions and nondeterminacy of results, and the necessary attempt to establish some order via synchronisation brings up dangers like deadlock. Real-time applications add to this the problem of temporal nondeterminacy (deadline over-runs, jitter) and use scheduling to ensure some measure of timeliness.

Indeed: In thread-based real-time programming, techniques are even more diverse and harder to assess than in other areas of concurrent programming. For a simple example, consider two periodic threads A and B with a common period and the additional requirement that (an instance of) B shall run only after (the corresponding instance of) A has finished so that B may use A's results. There is a bewildering range of possibilities (examples e) and f) are from [6]), neither of which is completely satisfactory:

a) concatenate A and B to obtain a single thread;

b) use static cyclic scheduling;

c) use wait-notify synchronisation;

d) turn B into an event handler whose execution is triggered on A's completion;

e) release B with an offset t relative to A where t is large enough to accomodate A and small enough so that B fits into the remainder of the period;

f) assuming fixed-priority scheduling, and that A is nonblocking, give A higher priority than B;

g) assuming fixed-priority scheduling with FIFO-within-priorities, and that A is nonblocking, give A and B equal priorities but release B with an offset t relative to A, where t is arbitrary but small.

Solutions a) and b) find it easiest to avoid threads for program composition even though they may be useful in conceptual modelling. Solution c) is what one learns in concurrent-programming textbooks under the head of "condition synchronisation". Since Java's wait/notify/notify-all complex is awkward to use and, in general, hardly analysable for temporal costs, Java's event mechanism (solution d)) seems to be the better alternative (cf. [17]) but introduces a linguistic and conceptual asymmetry into the problem. Solutions

f) and g) are what concurrent-programming textbooks admonish us not to do: "Never rely on scheduling for program correctness!". But since the real-time programmer has to rely on a scheduler for keeping his deadlines and therefore puts priorities into his code anyhow, the question is rather not one of principles but of software-engineering merits like clarity or maintainability. In examples f) and g), the programmer's intention and the correctness of his design are not easy to see, and the solution probably breaks if the software is altered later on. Solution e), on the contrary, makes the temporal succesion of A's and B's execution clear enough by fixing a separation point on the time axis. But such explicit timing has its disadvantage, too: when the program is moved to another machine, worst case execution times of A and B change and possibly require another choice for the offset value t.

In order to improve the software construction process and the analysability of its result, and to allow leaner or faster platforms, people have thought about restrictions of the language feature set and guidelines of use. In the Ada world, the Ravenscar profile ([5]) adopts fixed-priority preemptive scheduling with ceiling-locking as a well-understood basis for predictability and defines an appropriately reduced tasking model; more recently, a similar effort for Java was introduced in [17]. "Predictability" in this context means that a system can be guaranteed to satisfy temporal constraints, usually deadlines (cf. [8], ch. 1.3). The Ravenscar profiles are a major step forward, but cannot rule out other problems with threads like jitter or race conditions, and there is considerable research activity that explores the suitability of other models of concurrency, e.g. dataflow, for real-time programming (cf. [18]). An interesting branch in this activity is to move beyond working with the traditional release-time/deadline brackets and instead prescribe precisely when certain things should happen. Timing as a programming technique can avoid all the difficulties of threads and their synchronisation and achieve a degree of determinism where a sequence of sensor readings is mapped to a unique sequence of actuator settings, both in the value and the time domain. The price, as indicated in the comment to example e) above, is an inherent danger of overspecification. Time has many aspects and functions in software (and everywhere): a precise clock value may be relevant, but often we need less, e.g. a certain order of events, or that some action is repeated with a specified frequency. Time values that are not derived from the physical context but just brought in as a convenient way to coordinate concurrent activity may unnecessarily constrain or even prevent the realisation of a design.

We believe that restrictive efforts like the Ravenscar profile should be complemented by work that makes such new programming models available in Java, for experimentation or serious use. In the present paper, we describe the design and implementation of a Java framework for Giotto, a recent real-time language, which Henzinger, Horowitz and Kirsch designed for control software with periodic behavior [13].

Giotto distinguishes communications and computations. Communications occur at predefined times and are virtually instantaneous; they move values between sensors, computations and actuators. Computations, in contrast, take time and live in the intervals between the communications; computations use and produce values to realise control laws. – It may help to compare this layout with synchronous languages like Esterel, cf. e.g. [12], and with static clock-driven scheduling. In contrast to the first, Giotto's synchronous parts do not contain cyclic dependencies and therefore avoid problems with fixed points and consistency; furthermore Giotto realistically complements synchrony with non-zero-time computations. As regards static scheduling, note that a Giotto program defines a time-table for input/output events and not a schedule for the computational work; the former should reflect the temporal needs of the control problem, whereas the latter is indirectly constrained but not actually fixed by the communication instants: any scheduling scheme – clock-driven, preemptive fixed priority, earliest-deadline-first etc. – is welcome to try to map computational work to processor time so that the specified input/output times are kept. Giotto thus returns scheduling to its traditional role as part of the implementation: in this view, and contrary to what goes on in our examples f) and g), scheduling has no part in the actual programming but, by suitable allocation of resources, helps to make execution agree with the given program semantics. – A related programming model appeared in the TCEL language by Gerber and Hong [10], which features a distinction of temporally constrained "observable" events and "unobservable" computations, which, as far as data dependencies allow, can be placed anywhere on the time axis. This approach is somewhat more expressive and more complicated than Giotto; the main impetus of TCEL is to have a smart compiler rearrange code portions to improve feasibility.

To enable Giotto-style programming in Java, we build a framework called Jiotto, which can be used on any RTSJ compliant Java virtual machine. We will not bother with concrete Giotto syntax and, correspondingly, a translator, but rather define Java datatypes (classes), which represent abstract syntax, i.e. the elements of a Giotto program, and which the user instantiates and then passes to a likewise predefined executive component to get the specified behavior. Our approach has some similarity to the "embedded language" technique, which has become popular in the functional programming community (see e.g. [14]), because it allows the user to work in his accustomed environment and makes changes and additions easy. In our implementation we tried to keep to the Ravenscar Java profile but could not avoid some characteristic deviations.

We shall tell more about Giotto in Sect. 2. Section 3 will describe our framework building, and Sect. 4 will sum up and indicate future work.

2. A SHORT DESCRIPTION OF GIOTTO

To make our paper reasonably self-contained, this section recapitulates the main points about Giotto. Still, the reader may wish to consult [13] as the authoritative description. – Giotto is not a complete programming language but meant to define organisational skeletons that must be fleshed out with the help of some host language, e.g. with subroutines written in C for the computational work.

The principal elements of a Giotto program are ports, tasks, drivers and modes, which, loosely speaking, represent com-

munication infrastructure, work to be done, communication & control and configuration.

Ports function like program variables, with write and read operations and persistency between writes. Ports may be shared for communication. There are sensor ports, whose values come from the environment, and task ports and actuator ports for computed content.

Tasks compute values for their output ports from values of their input ports and then terminate. Giotto tasks must not be confused with threads but should be viewed as subroutines, or, even better, pure functions – without side effects, internal synchronisation points etc.. Tasks are invoked periodically, and privately used ports can give a notion of state across invocations; so there results some similarity to threads.

Drivers are responsible for task invocations. They load the input ports with values which may be constants or, as the general case, readings from sensor ports or task output ports. In addition, drivers can be guarded, i.e. evaluate a predicate on current port values and thus decide whether the task in question shall be actually executed or not. (So drivers can e.g. be used for the approximative technique of dealing with events in a time-driven context by polling.) Similarly, drivers are used for writing values to actuators.

A Giotto program consists of a nonempty set of modes, which represent distinct ways of system operation like e.g. take-off, cruising or landing for aircrafts. A mode definition names the tasks that shall be invoked and associates frequencies and drivers with them. Actuator updates are specified similarly. All frequencies are interpreted relative to a period which is assigned to the mode. – Mode switches are specified by a target mode and, again, a frequency and a driver for the decision making. There is one designated start mode.

Figure 1 shows a (meaningless) example similar to [13], p. 91 (concrete syntax for Giotto seems to be not fixed yet). Note that the definitions of driver-, guard- and task-functions are missing as they are programmed in some other language.

Next we describe the semantics of a Giotto program. For readibility and brevity, we use a semi-algorithmic style rather than give a mathematical definition based on sequences of states (but see [13]), and we leave out mode switching, i.e. assume that there is just one mode.

Let p be the period of the mode, and let task T be invoked with frequency f. Then the sequence $t_0 = 0, t_{i+1} = t_i + p/f$ defines the event times for task T; for actuators, the event times are defined in the same way. The ordered merge of all such sequences gives the event times of the mode. At any event time instant, several things must be done, stepwise and in a fixed order. The synchrony hypothesis here is that this work is not interrupted, and that the next event time is far enough in the future. As a further descriptive aid, we say that a task always is either active or not. A task, if active, ceases to be so at (the beginning of) its next event time; conversely, a task may become active only at (the end of) its event times. At the beginning, time $t = 0$, all tasks

```
inport
  port i0 type integer
  port i1 type integer
outport
  port o0 type integer
  port o1 type integer

task t0 input i0 output o0 function t0Function
task t1 input i1 output o1 funciton t1Function

driver d0 source o0 guard guardTrue destination i0
                                     function h0
driver d1 source o1 guard guardTrue destination i1
                                     function h0
driver d2 source o0 guard guardD2 destination o1
                                    function h0
driver d3 source o1 guard guardD3 destination o0
                                    function h0

mode m0 period 400 ports o0
  frequency 4 invoke t0 driver d0
  frequency 2 switch m1 driver d2

mode m1 period 400 ports o1
  frequency 1 invoke t1 driver d1
  frequency 1 switch m0 driver d3

start m0
```

Figure 1: Example Giotto program

are inactive, and all ports are suitably initialised. Now for all event times t of the mode, Fig. 2 describes the pertinent activities.

Let us emphasize two points. First, Giotto semantics prescribes when the input ports for a task invocation shall be loaded and when its results shall become available through its output ports. (And note that if several tasks have a common event time, the output ports of all of them are written before possibly read.) Second, Giotto semantics does not prescribe exactly when an invoked task must be executed, but only implies that execution must happen in a certain time interval, between the parameter passing at some event time t_i and the request for results at the next (for the task) event time t_{i+1}. There are several consequences:

- By the first remark, a Giotto program is outwardly deterministic. Depending on input values, processor load, platform characteristics etc., an invoked task will take more or less time to compute its results, but these results are made public for use always after the same period. Note that Giotto makes it easy to modify programs; since tasks are just functions without internal synchronisation points, the addition e.g. of some more tasks to a mode cannot lead to complications like deadlock that would make the specified temporal behavior logically impossible.
- By the second remark, there is much freedom for implementation, including internal nondeterminism. An obvious and easy scheme is to bind Giotto task

```
for all tasks T
  if t is an event time of T and T is active
    set T not active;
    write the result of T to its output ports;
  end if,
end for;
for all sensor ports S
  update S according to environment;
end for;
for all actuators A
  if t is an event time of A
    update A according to driver,
  end if;
end for;
for all tasks T
  if t is an event time of T
        and driver guard is true
    set T active;
    update input ports according to driver;
    arrange for T to be executed after t;
  end if,
end for;
```

Figure 2: Event Time Activities

functions to Java threads and apply rate-monotonic scheduling to them. In the next section, we describe how a master thread can ensure the required ordering of communications. If worst case execution times are known, schedulability analysis (per mode) can be done by the response test (cf. e.g. [7]), with task periods (= deadlines) explicitly given by the quotients of mode periods and task invocation frequencies in the program.

- Finally note that Giotto, like any language that allows to set temporal bounds for some activity, cannot on its own make the intended semantics come true: there must be a feasibility proof that combines program and platform characteristics. Even when such a proof is given, it may be wise to provide for deadlines that were missed because of exceptional conditions. Giotto, like the time-driven paradigm in general (cf. Kopetz' Time Triggered Architecture, [16]), provides an excellent basis for fault tolerance, esp. for error detection and containment: at the beginning of each event time, check that all task executions which should have finished did so indeed and produced acceptable results.

Let us conclude this section with a short note about mode switching. To preserve determinacy, Giotto forbids that more than one mode switch guard evalute to true at the same time. Another obvious problem is what to do with active tasks at mode switch time. Giotto semantics lets such tasks continue and tries to smooth things out by jumping to a point near the end of a round of the target mode. This is questionable; it e.g. may cause intolerable delay (think of the quintessential switch to failure mode). We are considering alternatives, including discontinuation of task evaluations, but at present have not worked out a good solution either.

(The current Jiotto version does not support mode switches that do not coincide with task terminations.)

3. THE FRAMEWORK

The goal is to enable the Java programmer to construct something similar to a Giotto program and have it executed in agreement with Giotto semantics. There also are some subsidiary "syntax-related" goals like checking wellformedness of the pseudo-program (e.g. that two tasks do not use the same port for their output), produce sufficiently abstract documentation or help with the analysis of temporal feasibility (relative to platform characteristics). Of these three, our framework at present contains only a serious attempt at the first item, sanitary checks.

3.1 Representation Technique

A Giotto program contains ports, tasks, drivers and modes; a mode contains task invocations, actuator-updates and mode-switches; the top-level notion, i.e. the proper program, is the set of modes together with an indication of the start mode. (Recall our informal explanation in Sect. 2 or consult [13] for more details). A Jiotto user has to create and assemble instances of these notions, according to his programming purpose. Most objects are generated directly from class definitions in our framework with the help of factory methods, using parametrisation to specify what exactly is wanted. A call like

```
createPort(InPort.class,0,42,''InPort-0'');
```

creates an input port object whose value type, as inferred (via overloading) from the initial value 42, is integer, whose identity number (for internal use, e.g.indexing) is 0 and whose string-identifier (for producing reports, error messages etc.) is "InPort-0". Objects of other "linguistic" types are created in a similar way, and sometimes additional setter-methods are employed for greater convenience or to break cyclic dependencies among to-be-defined objects. To create driver and task functions, the Jiotto user must provide his own class definitions, based on given interfaces. The interface for task functions e.g. contains just the relevant signature definition

```
public void f(InPort[] inPorts,
              ModePort[] modePorts,
              PrivatePort[] privatePorts)
```

Remember that such function definitions should be free of side effects, and if the function body creates objects, they must be allocated in scoped memory.

Internals of the classes for the Giotto elements – most of which are aggregates of some other – are not very interesting; we generally used arrays for collections because object creation is confined to an initial set-up phase of program execution.

The Java code in the appendix shows the small example of Fig. 1 in Sect. 2 as set up in `Jiotto`.

3.2 Execution Technique

Giotto semantics, as we already pointed out, is abstract enough to allow very different implementation strategies. One could e.g. try static scheduling and therefore expand the program-defined time table for task invocations (in the Giotto sense) into a more finely grained schedule that additionally fixes the actual intervals in which each task function is evaluated. Or one can choose some form of dynamic scheduling. In a RTJS and Ravenscar context, rate-monotonic fixed-priority scheduling seems an especially natural and easy choice, and we use this in our present implementation.

Recall that Giotto tasks are functions together with ports for argument/result values, and that these functions are evaluated repeatedly according to the frequency specified in the invocation part of a mode. As indicated above, we represent Giotto task functions by Java methods with a suitable signature, but to have the corresponding computational workload managed by a fixed-priority scheduler, we bind a Java carrier thread to each such method (Giotto task function) so that the run method of the thread repeatedly calls the function method. These carrier threads get rate-monotonic priorities. But we cannot employ standard technique to realise the periodic behavior. Remember that Giotto semantics, for each event time, demands a deterministic series of data movements that cuts across all tasks that finish/start at this time (Sect. 2). To secure the required order of reads and writes we have to get more explicit control over the concerned carrier threads. We introduce an additional control thread for this purpose. This unique thread has highest priority and is the only technically periodic thread (in the sense of calling `waitForNextPeriod()`); its period is defined by the coarsest temporal resolution possible for the present mode, i.e. the least common multiple of the invocation etc. frequencies. The carrier threads are directed by the control thread via wait/notify condition synchronisation on their event times and in this indirect way get their periodic behavior. This is a deviation from Ravenscar-Java but harmless since each wait/notify relates to just one pair of threads, viz. carrier and control thread, and does not create unpredictable wait sets. – The semantics of output ports also requires a moment of thought. They must be written only at event times and not change in between. Since in general task function results will be ready before the official time, one has to store them in auxiliary variables until then.

Another reason for introducing the control thread is the need to freeze tasks (and therefore carrier threads) when not used in some mode and later revive them again. – As an alternative technique, one may consider Java's event mechanism and use eventhandlers, which are schedulable objects too, for Giotto tasks. At present, we cannot offer experience with this possibility, and it may not provide much of an alternative as long as eventhandlers themselves are realised by threads.

The control thread implements the operational Giotto semantics described in Sect. 2, i.e. at any event time of the current mode it moves data around, makes decisions and releases tasks. It is responsible for some additional technicalities like, on a mode switch, adjusting carrier thread priorities (this is another unproblematic deviation from Ravenscar; one must do schedulability analysis mode-wise), determining its own period and so on. Furthermore, the control thread detects and acts upon deadline violations of Giotto tasks, though, at the moment, we have not defined and implemented proper fault tolerance measures. – Figure 3 shows our thread interaction structure.

3.3 Overall Structure

We follow the Ravenscar-Java philosophy of dividing the execution of a real-time application into two consecutive parts, the initialisation phase and the mission phase ([17]). In the non-timecritical initialisation phase a highest priority RealtimeThread is createted which in turn constructs all necessary objects (and threads) in immortal memory. Before terminating itself, this thread starts the application threads, which from then on run under the chosen scheduler policy, using only immortal or scoped memory. Ravenscar-Java offers an initialiser thread class whose run method has to be rewritten according to application needs.

In the case of Jiotto applications, this run method first creates the control thread and then constructs the pseudo-Giotto program, i.e. objects for ports, drivers, tasks and modes (and subsidiary objects for task functions and drivers); once again look at the code in the appendix. After termination of the initialiser thread, the control thread runs (we assume FIFO-within-priorities) and directs the execution of the carrier threads as described above. We found it convenient to add to the responsibilities of the control thread:

We let the control thread, which, by the way, is a singleton, act as the general interface that the Jiotto programmer uses for all his definitions and creations that form the pseudo-Giotto program; especially, the relevant factory methods for ports etc. are invoked through methods of the control thread object. When finally started, the control thread performs the sanity checks on the pseudo-program and does last arrangements for its execution, e.g. determines and sets carrier thread priorities.

4. CONCLUSION

We described a framework that supports the Giotto programming model in Java. This model is intended for periodic control applications, and, as far as this domain goes, has the virtue of combining the determinacy of the traditional cyclic executive with the flexibility of a threading approach: the Giotto programmer specifies the timing of I/O events but leaves it to the implementation to accomodate the computational work that realises functionality. Our implementation, which has a cyclic control thread supervise worker threads, nicely reflects this distinction.

Other implementation strategies are possible and possibly worthwhile, and we mentioned some minor alternatives in the text. A perhaps important improvement of control thread efficiency is this: In the present implementation, the control thread is periodic, relying on evenly spaced event times that are taken from the smallest common multiple of frequencies used in the present mode. This is likely to be wasteful; in a mode with invocation frequencies 3 and 7, there are only 9 event times, but our scheduler will inspect 21 for work to do. As an alternative, one can precompute an

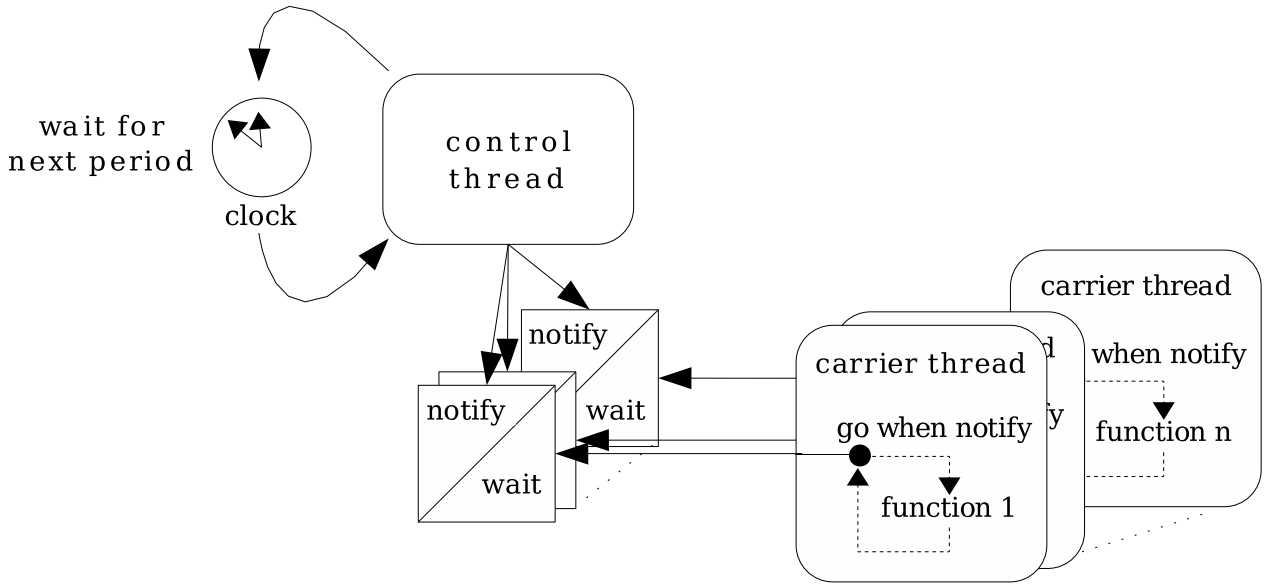

Figure 3: Thread Interaction Structure: periodic control thread and carrier threads

event list (as in discrete event simulation) that names event times and the associated work and have the control thread move from one event time in the list to the next. Since these times are not evenly spaced, we now need a timer and again deviate from the Ravenscar recommendations.

We shall combine the revision of our implementation with modifications of its scope:

First we want to integrate basic fault tolerance measures. If a carrier thread fails to produce its results in time, a reasonable strategy for the control thread is to use the most recent result instead. There are three possible causes for this failure situation: evaluation of the Giotto task function can produce a recognizably wrong result, stop because of a runtime error or take too much (possibly infinite) time. The first case requires some user-supplied healthiness test for results. In the second case the carrier thread should catch the exeception (i.e. not terminate but go on with its main loop!). The last case makes it necessary to abandon the current function evaluation; this of course means Asynchronous Transfer of Control, which is a problematic feature because it re-introduces nondeterminacy. Ravenscar excludes it; but it seems to be unavoidable in case of failure. – A related problem comes up with mode switches. As long as task functions really are pure functions, there should be no fundamental problem in abandoning an evaluation, and it may be better to substitute a dummy result than to delay the mode switch until a result from the old mode becomes available.

Second, we want to go beyond Giotto-style periodicity, which may become awkward or inefficient in practice. Consider an application where one task, every 500 ms, reads some value and adds it to a sum, and a second task, every 24 h, computes the mean value (which takes less than 500 ms too). In Giotto, periodicity offers three alternatives: first, a mean value becomes available only after another 24 h (which probably is useless); second, we release the mean value task every 500 ms too (doing nothing useful all day long); third, we throw in a mode switch every 24 h and back again 1 s later just to let the mean-value task execute once. Clearly, we need what in traditional terms is called "deadline smaller than period", and this should fit into the Giotto model. It only requires that event times need not necessarily bear read as well as write actions, and that task timings are defined by two numbers instead one, viz. frequency and relative deadline. – There is a somewhat more fundamental problem in the background: We mentioned in our introduction that the use of explicit timing to coordinate concurrent threads is in danger of overspecification; for something like the consumer-producer problem, it is natural to introduce a separating point in time, but a program should not specify a specific clock value for it, since the chosen value could be an unrealisable requirement while a whole range of other values would do the trick. So the question is whether one could specify the intended ordering with the help of time variables and solve for these unknowns in the context of schedulability analysis for a given platform, with known worst case execution times.

5. REFERENCES

[1] M. Amersdorfer. Jiotto – a Java framework implementing the Giotto semantics. Master's thesis, Universität Salzburg, 2004.

[2] P. Amey and B. Dobbing. High integrity ravenscar. In *Proceedings of Reliable Software Technologies – Ada Europe 2003*, volume 2566 of *Lecture Notes in Computer Science*. Springer, 2003.

[3] G. Bollella, B. Brosgol, P. Dibble, S. Furrer, J. Gosling, D. Hardin, M. Turnbull, and R. Belliardi. The real-time specification for Java. www.rtj.org, 2001.

[4] B. Brosgol and B. Dobbing. Real-time convergence of

Ada and Java. In *Proceedings of ACM SIG Ada 2001*, 2001.

[5] A. Burns, B. Dobbing, and G. Romanski. The ravenscar tasking profile for high integrity real-time programs. In *Proceedigs of Reliable Software Technologies – Ada Europe 1998*, volume 1411 of *Lecture Notes in Computer Science*. Springer, 1998.

[6] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the ravenscar profile in high integrity systems. Technical Report YCS-2003-348, University of York, 2003.

[7] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.

[8] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer, Boston, 2000.

[9] P. Dibble and A. Wellings. The real-time specification for Java – current status and future work. In *Proceedings of 7th IEEE Symp. on OO Real-Time Distributed Computing, ISORC 2004*, 2004.

[10] R. Gerber and S. Hong. Compiling real-time programs with timing constraint refinement and structural code motion. *IEEE Transactions on Software Engineering*, 21, No.5, 1995.

[11] H. Hagenauer, N. Martinek, and W. Pohlmann. Ada meets Giotto. In *Proceedings of Reliable Software Technologies – Ada Europe 2004*, volume 3063 of *Lecture Notes in Computer Science*. Springer, 2004.

[12] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, Boston, 1993.

[13] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time triggered language for embedded programming. *Proceedings of the IEEE*, 91(1), January 2003.

[14] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28A(4), December 1996.

[15] C. Kirsch. Principles of real-time programming. In *Proceedings of EMSOFT 2002*, volume 2491 of *Lecture Notes in Computer Science*. Springer, 2002.

[16] H. Kopetz. *Real-Time Systems*. Kluwer, Boston, 2001.

[17] Kwon, A. Wellings, and S. King. Ravenscar-Java: A high integrity profile for real-time Java. In *Proceedings of Joint ACM Java ISCOPE Conference 2002*, 2002.

[18] E. A. Lee. What's ahead for embedded systems? *IEEE Computer*, 33(9), September 2000.

[19] J. Ousterhout. Why threads are a bad idea (for most purposes). Usenix Technical Conference 1996, Invited Talk, 1996.

[20] A. Wellings. Is Java augmented with RTSJ a better real-time system implementation technology than Ada 95? *Ada Letters*, 23(4), December 2003.

APPENDIX

A. JIOTTO REPRESENTATION OF THE EXAMPLE GIOTTO PROGRAM

```
import jiotto.*;
public class JiottoExample extends Initialiser {
  public void run() {
    ControlThreadSingleton ct =
       ControlThreadSingleton.getReference();

    /* P O R T S */
    ct.setNumberOfInPorts(2);
    ct.setNumberOfOutPorts(2);
    InPort i0 =
      (InPort) ct.createPort(InPort.class, 0, 0, "i0");
    InPort i1 =
      (InPort) ct.createPort(InPort.class, 1, 0, "i1");
    OutPort o0 =
      (OutPort)ct.createPort(OutPort.class, 0, 0, "o0");
    OutPort o1 =
      (OutPort)ct.createPort(OutPort.class, 1, 0, "o1");

    /* D R I V E R S */
    ct.setNumberOfDrivers(4);
    TaskDriverGuard guardTrue = new GuardTrue();
    ModeDriverGuard guardD2 = new D2Guard();
    ModeDriverGuard guardD3 = new D3Guard();

    TaskDriver d0 = (TaskDriver) ct.createDriver(
         TaskDriver.class, 0, guardTrue, 1, "d0");
    d0.transfer(o0, i0);

    TaskDriver d1 = (TaskDriver) ct.createDriver(
         TaskDriver.class, 1, guardTrue, 1, "d1");
    d1.transfer(o1, i1);

    ModeDriver d2 = (ModeDriver) ct.createDriver(
         ModeDriver.class, 2, guardD2, 1, "d2");
    d2.transfer(o0, o1);

    ModeDriver d3 = (ModeDriver) ct.createDriver(
         ModeDriver.class, 3, guardD3, 1, "d3");
    d3.transfer(o1, o0);

    /* T A S K S */
    ct.setNumberOfTasks(2);

    TaskFunction t0Function = new T0Function();
    Task t0 = ct.createTask(0, t0Function, "t0");
    t0.addInPort(i0);
    t0.addOutPort(o0);

    TaskFunction t1Function = new T1Function();
    Task t1 = ct.createTask(1, t1Function, "t1");
    t1.addInPort(i1);
    t1.addOutPort(o1);

    /* M O D E S */
    ct.setNumberOfModes(2);

    Mode m0 = ct.createMode(0, 400, "m0");
    m0.setNumberOfModeSwitches(1);
    m0.setNumberOfTaskInvocations(1);

    Mode m1 = ct.createMode(1, 400, "m1");
    m1.setNumberOfModeSwitches(1);
    m1.setNumberOfTaskInvocations(1);

    m0.addModeSwitch(m1, 2, d2);
    m1.addModeSwitch(m0, 1, d3);

    m0.addTaskInvocation(t0, 4, d0);
    m1.addTaskInvocation(t1, 1, d1);
```

```
    /* S T A R T */
    ct.setStartMode(m0);
    ct.go();
    try {
      ct.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    new JiottoExample().start();
  }
}
```