Arranging Partitures for Models and Theorems

Johannes Helander and Margus Veanes
Microsoft European Innovation Center, Ritterstrasse 23, 52072 Aachen, Germany
Microsoft Research, One Microsoft Way, Redmond, Washington 98052
jvh@microsoft.com, margus@microsoft.com
September 30, 2008 (Updated March, 2009)

ABSTRACT

In order to do reliable and efficient real-time scheduling of a software application or *component, it is necessary to know as much as possible of the temporal behavior of a program up front. To make it possible to compose schedules it is necessary to know about the temporal behavior of the various applications and components involved. We describe the behavior using a Partiture, a description of the temporal phases of a program and their interdependencies. We then treat the partiture as a graph and model its semantics using abstract state machines. We then convert the state machine to a set of constraints and use the Z3 theorem prover to validate and solve the schedule. Finally the output is converted to a partially instantiated schedule, which can be executed on hardware, ranging from 8-bit microcontrollers to multicores and distributed systems. This work-in-progress paper shows how a number of common execution patterns map to a graph and model and how this interfaces with the theorem prover. It introduces an improved semantic description of partitures and for the first time shows how satisfiability modulo theory (SMT) could be used to perform scheduling analysis, with the practical side demonstrated by a simple case study. This sets the stage for planned future extensions to multiple resources and multicore.*

INTRODUCTION

To go from a program implementation and specification to an actual runtime execution that fulfills the constraints specified, we need a tool chain that consists of three parts: 1) discovering what an application or component does and what it should do; 2) figuring whether the program or a combination of programs can meet the specification and optimizing for specific requirements creating a schedule; and 3) executing the resulting schedule on actual hardware in a way that adheres to the validation of the previous step.

The tool chain is analogous to a compiler, where the front end converts the program to an internal representation, which is then analyzed and optimized by the middle, and finally the back end generates code that is executed within a known runtime. The main difference is that a regular compiler deals with functional properties of the program, while the temporal behavior and the corresponding schedule is non-functional.

The partiture describes temporal behavior

We represent the intermediate states of the temporal behavior of the program as a *partiture*. It is analogous to the short score in music, also called a partiture, that specifies when and what individual instruments should be playing without dealing with how the first violin is supposed to implement the sounds or other details relevant only to that instrument. A partiture is a collection of scores, each consisting of a number of bars. Each score corresponds to a single activity and each bar corresponds to a single temporal phase of a program. The bar is the time where a note is played, or in our case where a phase of a program is executed without caring what exactly is played (determined by the notes in the bar or the implementation of a component). The bar in essence is a single time constraint, determining when a note is played and for how long. More concretely we define it to have an earliest start time, a deadline, and a duration estimate. The duration is estimated by WCET analysis, stochastic methods based on instrumentation, or by programmer input. The quality of the estimate largely determines the hardness of the resulting schedule.

In [2] we show how partitures could be generated by static analysis and tracing from existing concurrent programs. We have also experimented with using instrumentation and stochastic estimation to derive timing information from a program running in a potentially chaotic non-real-time environment. The partiture is represented by a domain-specific language and in [1] we use XML syntax.

The temporal requirements of a program still need to be specified. This can be done through a domain-specific language or any formal definition.

The middle part of the time compiler validates, optimizes, and composes schedules

The partiture can be converted to a graph. In [2] we looked at the topology of the graph and optimized the schedule for sequential execution on a microcontroller. In [1] we defined a model for part of the partiture semantics. However, the semantic definition we have presented earlier, while sufficient to describe a number of scenarios, is incomplete in the sense that it does not capture all or most relevant execution patterns and is not quite precise enough for formal analysis. In this paper we examine the relevant scenarios in more detail and enhance the semantic definition further. While this may not be the ultimate definition we are now attempting to make it precise enough that we can feed the graph into a theorem prover.

We describe the semantics of the partiture graph as an abstract state machine based model program. The ASM essentially describes all possible valid schedules. If there are any valid schedules then the partiture is valid. If there are multiple valid schedules then some of them may be better than others. We attempt to find a single valid schedule by converting the partiture turned model program together with the partiture definition, platform constraints, and external constraints into a theorem using a process described in [5]. The result, if any, is again a valid schedule. The theorem prover does not have a concept of optimality. Instead it is possible to add further constraints, which may or may not be achievable. An iterative process can be used to use e.g. a binary search to find the *best* schedule. For instance to optimize for speed if no solutions were found, double the allowed time, if solutions were found decrease it by a quarter.

The Z3 theorem prover [4] does not only answer by saying it has found an answer but it will also specify what it is. If the question is well constructed the result is meaningful and can be converted to a (partially) instantiated schedule, which is used at runtime.

The runtime executes the schedule

The resulting schedule depends on the platform constraints. On a single microcontroller without preemption the schedule becomes a fixed table. An example of how this can result in a very efficient but nonflexible implementation is demoed in [3] where an XML web server with a real-time scheduler and drivers could run in under 4KB ROM and 128 *bytes* of RAM. Here the schedule is a predetermined table. In the demo the values in the table were manually calculated. In this paper we attempt to automate the process.

On a multicore, in a distributed system, or in a flexible system the schedule would be a partial evaluation, leaving flexibility to the runtime scheduler. The stochastic scheduler in [1] goes to an extreme and changes the estimates in the bar during runtime, yet an initial admission check is still required, which is difficult to do in a resource-limited system in real-time, considering that the scheduling is after all fundamentally NP-hard. Thus an offline, or semi-offline service based approach would be appropriate even here.

DEFINITION OF PARTITURE GRAPH

The partiture and corresponding graph consist of bars. Each bar represents a temporal phase of a program. The bars (nodes) are linked by edges that depict dependencies between bars. A directed edge is a causality constraint, meaning one bar must complete before the other can start. A bar represents one time constraint [**Error! Reference source not found.**] and some resource requirements (e.g. CPU, memory, bandwidth) during the time. To make the definition more concise we make it recursive.

Partiture := set of scores
Score := Nodes connected by Edges

Node := Score | bar | choice
Bar := labeled time constraint = <label, earliest start time, deadline, duration, resources>
Edge := directed causal constraint | undirected exclusion constraint
Choice := set of scores representing *alternative* execution

A node can be in a number of states: Waiting, Enabled, Expired, Completed.

Each node can have multiple outgoing edges, representing parallel causality. The outgoing edge points to either another node inside or outside the current containing node or to the end of the current containing node. The outgoing edges are triggered when the node has completed (in the nested case when the edge leading to the node completion has been triggered). Each node can have any natural number of incoming edges. Depending on the node the incoming edges can be either treated as OR or as AND. The nodes with no incoming edges are enabled when the containing score is triggered. AND nodes are enabled when all incoming edges are triggered. OR nodes are enabled when any incoming node has been enabled.

In addition to directed edges the graph can contain undirected edges, representing a mutual exclusion constraint. Two nodes connected by an undirected edge can never be enabled simultaneously, facilitating transactions and commutative operations.

Remote dependencies are represented by a node that uses network bandwidth (message transfer) and a directed edge through it from the sender to the receiver. A further limitation is that no edges can pass between remote nodes without going through a message transfer node. Thus undirected edges are strictly local and no instant messageless triggers are possible remotely.

In [2] we explored sequentialization of execution by use of bars. In that paper we used red directed edges to depict data dependencies and blue edges to depict execution order within a thread. Another way to look at this is to say that blue edges have two additional constraints compared to normal (red) directed edges: they cannot trigger remotely as a message transfer node is not present; and in order to schedule other bars between the nodes the target platform must support preemption.

The goal of the scheduling tool is to convert a partiture into a schedule.

Schedule := a sequence of (partially) instantiated bars for each CPU

In other words the goal is to flatten a potentially complex partiture to a simple sequence, one for each processor. There may be multiple processors either due to multicore or due to multiple machines. Other resources, such as network bandwidth or memory, are scheduled implicitly by scheduling the computation that allocates (frees) memory or sends (receives) messages.

REPRESENTING EXECUTION PATTERNS AS GRAPHS

Figure 1: Simple sensor to actuator

Consider a simple robot controller that reacts to changes in the physical world by first reading a sensor value, analyzing it, and finally writing some computed value to an actuator. The score for such a scenario is depicted as a graph (Figure 1) and as an XML description (Figure 2). This is also the score we model and solve using a theorem prover later in this paper.

While the score may seem simple it actually hides some implied complexity that we will now disseminate further. First, the pipeline needs to be executed multiple times. In [1] we argue that it is better to keep the loop in the scheduler than the program. Figure 3 depicts the score with a loop added, along with a sensor to actuator and an inter-iteration time constraint.

```
<score name="hardsample">
  <bar name="read" duration="PT0.002S"
       slack=”PT0.0005S”>
    <repeats count="1000" offset="PT0.02S"/>
    <trigger name="filter" />
  </bar>

  <bar name="analyze" duration="PT0.001S" />
    <trigger name="output/>
  </bar>

  <bar name="output" duration="PT0.001S">
  </bar>
</score>
```

Figure 2: Sensor to actuator as XML

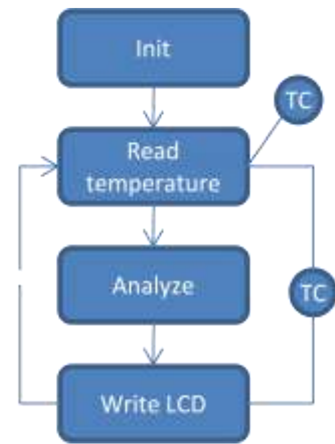


Figure 3: Sensor to actuator with more detail. TC = Time Constraint.

The sensor to actuator time constraint defines the maximum allowed time between reading and writing the physical environment. The inter-iteration time constraint defines how often the sensor should be read. Both become constraints in the model and SAT solver. If the loop is unrolled the score looks like figure 4, which depicts the semantics of the score. The resulting schedule could be repetitive, however, when the score is composed with other scores the loop may have to be partially unrolled to produce a valid scenario. In the unrolled graph all bars are initially not enabled. When the score is enabled the Init with no incoming edges will become enabled and it in turn will enable the rest. It can be seen from the graphs that all joins are OR joins.

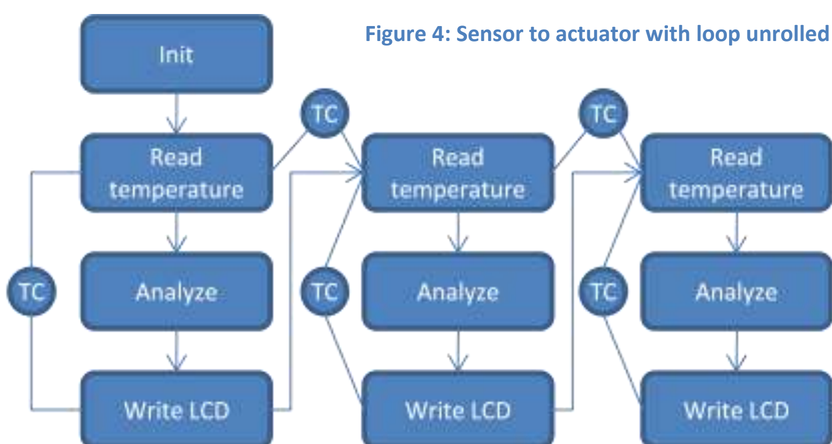
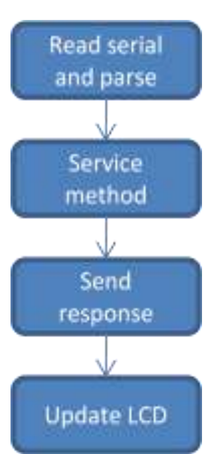


Figure 4: Sensor to actuator with loop unrolled

When there are multiple sensors and actuators the score would look like figure 5. The time constraints from each sensor to each actuator are omitted in the drawing. This score highlights an AND join. Here a pull-up can be performed only when there is no stalling but the ground is getting close. It goes without saying that the author would not like to be in this airplane, a real control algorithm would be far more complex.



A simple web 2 server running on a microcontroller could look like figure 6. The last optional bar displays the result on the LCD. A server would process multiple messages. For instance in the demo we showed in [3] the messages can either result in an Add method or a Subtract method. While these two are remarkably similar, sometimes it may be useful to include explicit scores for alternative executions. A score with alternative bars for Add and Subtract are depicted in figure 8.

Figure 6: Simple XML web server

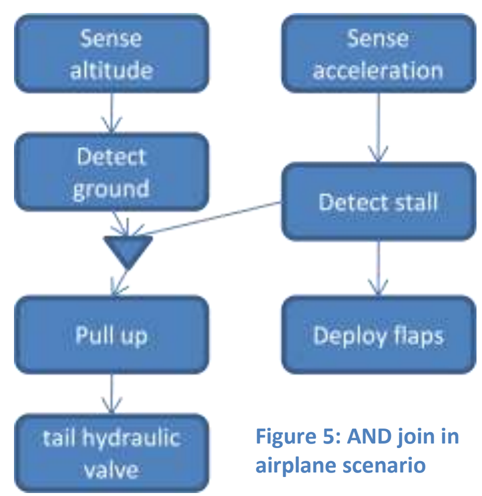


Figure 5: AND join in airplane scenario

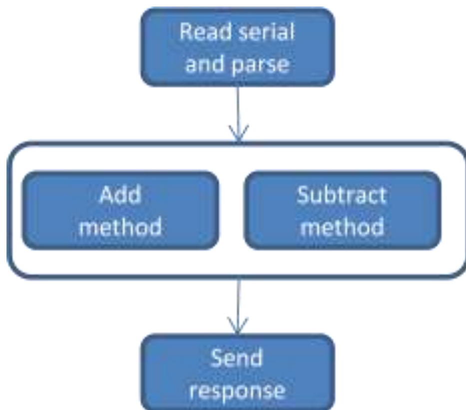


Figure 8: Web server with choice bars

A web server that listens to multiple inputs then dispatches to multiple alternative methods and finally responds to alternative destinations is depicted in figure 9.

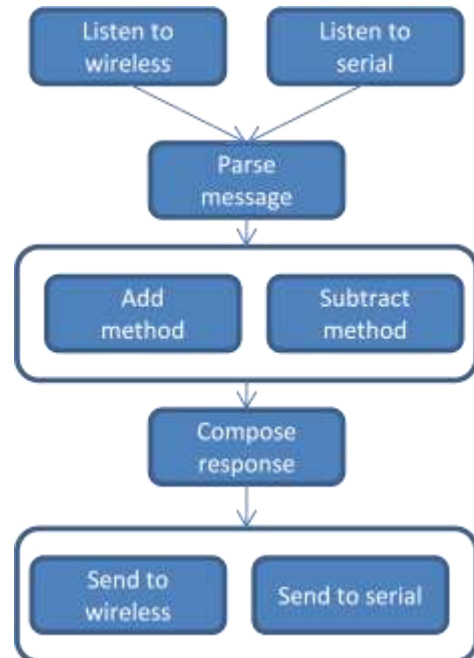


Figure 9: Web server with multiple inputs and choice bars.

Potentially parallel execution is depicted in figure 10. The video and audio processing are independent and can be done either in arbitrary sequence or in parallel, depending on resources available on the target machine. The partiture always expresses potential parallelism. As necessary it can be sequentialized either to a completely or to a partially sequential schedule, or executed in parallel when enough cores are available. The AND join prevents the downstream code from executing before both audio and video processing has completed.

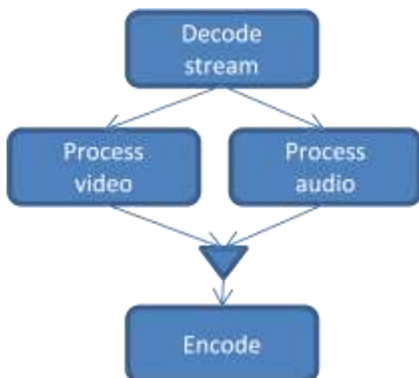


Figure 10: Parallel bars.

Figure 11 is a remote procedure call. The message transmission is a bar that estimates the time taken for the message delivery and resources the required network bandwidth. Only message transmissions and time constraints can span multiple machines.

Figure 11: Remote procedure call.

Figure 12 depits software where media data is handled by multiple threads, each doing a phase in the processing. While it is debatable whether this is the best design it should at least be possible to describe its temporal behavior. All the joins are OR as the pipeline will be active whether there is data from various sources or not.

Figure 12: Audio processing pipeline.

Figure 13 depicts a mutual exclusion constraint for the purposes of transactions. A transaction is wrapped into a separate score that is then excluded with other transactions on the same object. This simply adds a constraint to be solved by z3. Finally, figure 14 depicts composition, where two separate scores coexist in parallel. This is simply a combination of their set of constraints that are adhered to when scheduling.

Figure 13: Mutual exclusion constraint.

Figure 14: Composition of two independent scores.

MODELING EXECUTION

We model the valid executions of partitures as traces of an abstract state machine (ASM) based *model program*. The model itself is composed by the types of constraints that they define. The timing model (figure 15) deals with the timing of bars, the earliest start time, deadline, and estimated duration (e.g. WCET). The values do not need to be set for all bars. Each resulting schedule will set them to some valid value. The model describes all valid executions on a single processor machine (assuming bars are not preempted). The model describes a sequence and inter-bar synchronization is an indirect result of the bars waiting for the predecessor to complete.

```
namespace Timing
 enum Bar A B C

 var t as Integer = 0
 var earliest as Map of Bar to Integer = {A -> _, B -> _, C -> _}
 var deadline as Map of Bar to Integer = {A -> 2, B -> 4, C -> 6}
 var duration as Map of Bar to Integer = {A -> 2, B -> 1, C -> 1}

 [Action]
 Do(b as Bar)
  require t >= earliest(b) and t < deadline(b)
  t := t + duration(b)

 [Action]
 Skip(dt as Integer)
  require dt >= 0
  t := t + dt
  t := t + dt
```

Figure 15: *Timing* model with partially fixed sample

```
namespace Trigger
 var bars as Set of Bar = {A,B,C}
 var graph as Set of (Bar,Bar) = {(A,B),(B,C)}

 [Action]
 Do(b as Bar)
  require b in bars
  require not exists e in graph where Second(e)=b
  bars := bars - {b}
  graph := {e | e in graph where First(e) <> b}
  graph := graph - {(b,b+1)}
```

Figure 16: *Trigger* model with sample bars

In a manycore machine that has unlimited numbers of cores, all parallelism can be fully exploited. However, the synchronization mechanism of the single processor does not work. Instead a time triggered schedule is possible, where the earliest start time of the follower >= deadline of predecessor. In a multicore with a limited number of parallel cores a hybrid of the two models we have explored so far is needed, including a combination of the synchronization models. We are working on the correct formulation.

The causality constraints are modeled in the trigger model (figure 16). The correct model formulation uses a comprehension or a bounded quantifier to describe the relationship (in comments). At the time of the submission of this paper we are in the process of fine-tuning the conversion of complex comprehensions to theorems and are limited to simple graphs for the time being.

Adding memory budgeting to the scheduling is simple. In the timing model, add m := m + mem(b) after t:= t + duration(b). If mem(b) is the integer value of the memory allocated minus freed in the bar then m will keep track of the memory through any schedule. Like with the global time t, the memory m can be constrained by a global invariant. A memory budget is thus alike a time constraint (itself a time budget). The global constraints will limit the search space and thus possible schedules. The composition of models themselves limit the search space, the state space is the intersection of the composed models.

USING THEOREM PROVING

For theorem proving based bounded reachability analysis, a model program is viewed as a symbolic transition system [4] over a background *T* (including linear arithmetic). Given is an upper bound on the number of steps, given by the total number of bars, and a reachability condition that represents the state where all bars have been scheduled. The given model program, together with the bound and the condition, are translated into a formula φ such that φ is satisfiable in *T* iff a valid schedule exists. Provided that φ is satisfiable, a valid schedule is a sequence of bars that is represented by a valid action trace of the model program. Such an action trace can be extracted from a model for φ. Recent advances in *SMT* based theorem proving techniques, that is a middle ground between

pure SAT solving and full first-order theorem proving, makes this approach feasible for expressive fragments of *T*. Instead of encoding the verification task as a propositional formula the task is encoded as a quantifier free formula. The SMT based bounded reachability analysis of model programs is described in [4,5] and is based on the SMT approach to bounded model checking introduced in [9]. We use the SMT solver Z3 [10]. Z3 supports equality reasoning, linear arithmetic, fixed-size bit-vectors, sets and arrays.
Given a model program P a quantifier free reachability condition ϕ and a step bound k, a formula φ of size $k\cdot(|P|+|\phi|)$ is constructed as an input to z3. If the formula is satisfiable, a satisfying solution provides an action trace of length k from an initial state to a state satisfying ϕ. Such an action trace corresponds in our case to a fixed schedule.

TURNING THE RESULT INTO A SCHEDULE

Running the model through the analysis tool creates input for z3, which then produces a valid answer if there is one. The answer is converted by the tool back to a model trajectory in form of an action trace. Using the partiture in figures 2 and 3 and the models in figures 15 and 16, the trajectory is **Do(A),Skip(1),Do(B),Do(C)** as a solution to the reachability analysis of the composed model **Timing*Trigger** with the reachability condition **bars = {}** and a step bound 4.

The z3 output is longer because it includes the full model, not just the action trace, see Figure 17. At this point it is too early to make statements about the performance, however z3 is being used to solve hundreds of thousands of simultaneous constraints in seconds, in the context of program analysis.
The resultant model trajectory representing the schedule can be converted to a table appropriate for the runtime

```
partitions:
*0 -> true
*1 -> false
*2 {t0 action0 action1 action2} -> 0:int
*3 {earliest0 earliest1 earliest2 earliest3 earliest4} -> {*10 -> *2; *6 -> *2; *8 -> *2;
else -> *4}
*4 {Undef_Integer} -> 7:int
*5 -> {else -> *4}
*6 {Do_00 action3 Skip_03} -> 1:int
*7 -> {*6 -> *2; else -> *4}
*8 {Do_01 t1} -> 2:int
*9 -> {*8 -> *2; *6 -> *2; else -> *4}
*10 {t2 Do_02 Do_03} -> 3:int
*11 {deadline0 deadline1 deadline2 deadline3 deadline4} -> {*10 -> *14; *6 -> *8; *8 -> *10;
else -> *4}
*12 -> {*6 -> *8; else -> *4}
*13 -> {*8 -> *10; *6 -> *8; else -> *4}
*14 -> 6:int
*15 {duration0 duration1 duration2 duration3 duration4} -> {*10 -> *6; *6 -> *8; *8 -> *6;
else -> *4}
*16 -> {*8 -> *6; *6 -> *8; else -> *4}
*17 {bars0} -> {*10 -> *0; *6 -> *0; *8 -> *0; else ->
*18 {bars3 bars4} -> {else -> *1}
*19 -> {*6 -> *0; else -> *1}
*20 -> {*6 -> *0; *8 -> *0; else -> *1}
*21 {graph0} -> {*25 -> *0; *23 -> *0; else -> *1}
*22 {graph2 graph3 graph4} -> {else -> *1}
```

Figure 17: Part of z3 model of the above

environment (e.g. a C table of structs). This table is then executed by the scheduler, which can be quite simple. We previously demoed using a hand compiled schedule on an AVR 8-bit microcontroller [3]. The system implemented a small XML web service over a serial line with futures organized along the partiture in figure 7.

RELATED WORK

Graph based scheduling analysis was previously presented by the author in [2]. The bars in the partiture correspond to time constraints, described in [6]. Constraint based scheduling is itself an extension of earliest deadline first (EDF). Time constraint scheduling has later also been called logical execution time [8]. The partiture

[1] is a description of temporal behavior, separate from the functional implementation. A limited form of a separate temporal virtual machine was first described in Giotto [5]. This was concept was generalized by the author to distributed computing [12], concurrent programs [2] and scalable execution together with futures [1]. This paper further expands that work.

Time triggered architecture, one possible simple approach for manycore scheduling has been well researched, e.g. [11]. A visual grammar for describing temporal behavior together with system composition has been described e.g. in [13]. It can be used to describe time for data flow in media processing. The graph described in this paper can be used for media processing but is more general, with the obvious cost of being more difficult to process.

Reachability analysis of model programs has been described in [4][5]. To the best of our knowledge, this is the first paper where using SMT for graph based schedulability analysis has been proposed.

CONCLUSION

We presented ongoing work in creating a comprehensive tool chain for scheduling and analysis of temporal properties in complex concurrent programs. A hierarchical graph was introduced for representing partitures, the temporal behavior of components and applications. The graph depicts time, causal, topological, and commutative constraints. Examples of how the graph can represent common programming patterns were shown.

The graph was validated and serialized to a schedule using abstract state machines and a state of the art SMT solver. While the fully general solution is still work in progress, we were able to show how to schedule a simple partiture sequentially for a single CPU together with memory requirements.

We believe this work can lead to a better understanding and methods for scalable scheduling for concurrent programs, including distributed and parallel execution. Validation and partial evolution is done at (re)configuration time, leaving the simpler problem of correct implementation of the schedule to runtime.

BIBLIOGRAPHY

1. **J. Helander, R. Serg, M. Veanes, P, Roy.** *Adapting Futures: Scalability for Real-World Computing,* RTSS 2007.
2. **S. Mohan, J, Helander.** *Temporal Analysis for Adapting Concurrent Applications to Embedded System*s, ECRTS 2008.
3. **R. Serg, J. Helander**. *Using XML Web Services for Embedded Systems Interoperability. World's Smallest Web 2.0 Server Demo*. Demo session at Pervasive 2008, Sydney, Australia, May 2008.
4. **M. Veanes, N. Bjørner, A. Raschke.** *An SMT approach to bounded reachability analysis of model programs*. In FORTE'08, LNCS. Springer, 2008.
5. **M. Veanes, A. Saabas**. *On Bounded Reachability of Programs with Set Comprehensions*. In LPAR'08, LNCS, 2008.
6. **J. A. Stankovic and K. Ramamritham**. *The Spring Kernel: A New Paradigm for Real-Time Systems*, IEEE Software, May 1991.
7. **T. Henzinger, C. Kirsch, and S. Matic.** *Schedule carrying code*. In Proceedings of EMSOFT, Philadelphia, USA, October 2003. LNCS 2855, pp. 241--256, Springer, 2003.
8. **E. Farcas, C. Farcas, W. Pree, and J. Templ**,*Transparent distribution of real-time components based on logical execution time*, SIGPLAN Notices, Volume 40 Issue 7, pp. 31-39, 2005.
9. **L. de Moura and H. Ruess and M. Sorea**. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. Proceedings of the 18th International Conference on Automated Deduction (CADE'02), LNCS 2392, pp 438—455, 2002.
10. **L. de Moura and N. Bjørner**. *Z3: An efficient SMT solver*. In Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'08), LNCS. Springer, 2008.
11. **H. Kopetz and K.H. Kim.** *Temporal Uncertainties in Interaction among Real-Time Objects*, IEEE Computer Society's 9th Symp. on Reliable Distributed Systems, Huntsville, AL, Oct. 1990, pp.165-174.
12. **J. Helander, S. Sigurdsson**, *Self-Tuning Planned Actions: Time to Make Real-Time SOAP Real*, Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Seattle, May 2005.
13. **O. Sokolsky.** *Performance Analysis of AADL Models Using Real-Time Calculus.* Presentation at Monterey Workshop. Budapest, Hungary. September 24-26, 2008