

# SafetyChip

## A Time Monitoring and Policing Device

Gustaf Naeser  
Dept. of Computer Science  
and Electronics  
Mälardalen University, Sweden  
gustaf.naeser@mdh.se

Lars Asplund  
Dept. of Computer Science  
and Electronics  
Mälardalen University, Sweden  
lars.asplund@mdh.se

Johan Furunäs  
Dept. of Computer Science  
and Electronics  
Mälardalen University, Sweden  
johan.furunas@mdh.se

### ABSTRACT

The SafetyChip proposes a strategy where parts of the effort invested in the formal verification during the development of a system can be reused during the system's operation.

The strength in a formal verification of a system is that a system can mathematically be proven to fulfil certain requirements, e.g., timing requirements. The SafetyChip uses information from verification to monitor and police a system during run-time. The monitoring is done by surveillance of the applications communication with the run-time kernel. If deviance from the predefined verified behaviour is detected, the SafetyChip can signal (police) this in different ways, e.g., by generating interrupts the system can respond to.

In our experiments we use systems written in Ravenscar compliant Ada code and have automated model extraction from source code to the models used to verify the system.

This paper presents the functionality and design of the SafetyChip. Properties of an implementation of the SafetyChip are also presented.

### Categories and Subject Descriptors

B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance

### General Terms

Reliability

### Keywords

Hardware monitoring, Kernel, FPGA

## 1. INTRODUCTION

Safety has been an important issue for a very long time in the design of complex systems, where a malfunction can result in a severe catastrophe. Examples are the level A software in an air-plane or the software to control the rods

in a nuclear power plant. These applications are very sophisticated and can afford triple redundancy, and cope with the increase in complexity of the total system.

At the same time as more and more is being controlled by computers, the awareness of the risk for malfunctions in the system has increased. Several application areas will have to consider safety a lot more in the future. The automotive industry does not have to certify their systems yet, industrial robotics can today rely on electronic fences, but in the future these robots may work side-by-side with humans, and thus require a much higher degree of safety.

There are several principles that can be used to ensure safeness for these systems. One of the major issues is an occurrence of situations not envisioned during the system development. Such a malfunction can for instance be due to radiation; a single bit flip can cause a single processor system to fail. Many safety-critical applications are also hard real-time applications, i.e., systems where the result must be delivered in time as well as being correct. Although these kinds of systems have been excessively tested there is a risk that the actual worst case execution times have not been detected, i.e., the execution time of a calculation can take longer than estimated and thus make the system enter an unexpected system state. This paper assumes that the execution times for sequential code is known.

The most common way to get a higher level of safety to count for situations that are out of control for the software in a computer system, apart from excessive testing, is to use double or triple redundancy. This has of course the wanted effect to reduce the effect of a single bit flip and other events, since it increases the system's safety by ensuring that the safety relies on the correct operation of the majority of components rather than one single component. The main problem, however, with software systems is the inherent complexity. High complexity makes it infeasible for a human to fully grasp or test the system and hence the operation of the full system will not be fully known. Adding redundancy to a given software system will further increase its complexity with the added number of software components. Another drawback is the cost. It is more expensive to build a double or triple redundancy system.

A solution to this dilemma is to use methods that can both ensure that the system will perform as required, and after a system has been verified to use methods similar to a compiler, i.e. the power and strength in a computer is used to generate the required supervising system (in the case of a double redundancy system). Furthermore if the supervising system or monitoring system is implemented in hardware

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

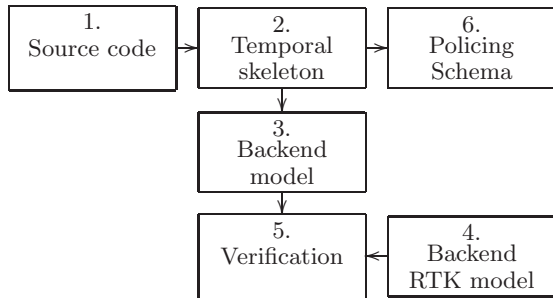
SIGAda'05, November 13–17, 2005, Atlanta, Georgia, USA.  
Copyright 2005 ACM 1-59593-185-6/05/0011 ...\$5.00.

instead of software; the software complexity is not increased. A possible benefit gained with this approach is the reduction in production cost compared to a system developed using redundancy.

A desirable alternative to this is to ensure the flawless operation of a single processor by adding a monitoring and policing component. There are several benefits from this solution, e.g., the complexity of the function performed by the processor might be kept down as the number of components can be held low. The SafetyChip proposes an architecture with foundations in pre-run-time formal verification and run-time monitoring.

Ada, with the Ravenscar tasking profile [1], is well suited for the development of high-integrity systems [2] and efforts to verify Ada systems are described in, e.g., in [3,4]. These works have focused on the ability to verify that properties of a system satisfied by the systems specification or implementation, pre-run-time. However, once the system is running any links to the formal methods and the verification are discontinued. The SafetyChip proposes a strategy where parts of the formal verification can be used to monitor and police the execution of the final system. A strategy like this should be attractive since it reuses the efforts made when formally verifying the system.

The SafetyChip implements a combination of strategies to accomplish its task. The development flow for a system using the chip is shown in Figure 1.



**Figure 1: A framework for formal verification of source code.**

The development starts with the source code of a system. The code is initially transformed, using automated model extraction, into a temporal skeleton which represents the timing behaviour of the code. The skeleton is formally verified together with a model of the real-time kernel to be used in the final system. When the verification is completed the skeleton can be said to represent the desired timing behaviour of the system. If the behaviour of the skeleton, and hence the code, is not the desired one the source will have to be changed and the verification started over again. When a skeleton passes verification the schema the SafetyChip should monitor can be distilled from its skeleton. The source code is compiled and, the RTK is synthesised and the two are linked into a running system together with the chip.

## 2. SYSTEM MONITORING

One way to gain knowledge of the run-time operation of a running system is through monitoring. During the system's execution the monitor observes and records a selected set of events, called the observed events.

The observed events can be either native system events that can be observed somewhere in the system, or they can be events generated by special purpose instructions that are injected into the system. Injected code can, e.g., trigger interrupts or output data. The drawback with injected code is that it changes the system's execution by adding, at least, the execution time needed by the injected instructions. Observation of native events does not tax the system in the same way but it requires that the monitor itself is more intimately connected with the tap point for the events. In the case where external events are generated the hardware of the monitored system need not be changed at all and a monitoring device can be fully implemented off-chip, even in another computer. When native events are observed the hardware needs to be extended or adjusted to allow monitoring.

The traditional architecture has been an off-chip design using a wiretap of the buses and registers to extract the monitored events. However, with the developments in System on Chip (SoC), it becomes more attractive to include the monitoring hardware in the system itself, on-chip [5,6,10,11]. On-chip solutions also have the advantage that they are easier to make non-intrusive, i.e., they do not change the run-time behaviour of the system they monitor.

### 2.1 Capture Rate

A problem that has to be addressed in any monitoring system, off-chip or on-chip, is the highest rate at which the monitor can capture and process observed events, which will be denoted as the monitor's *capture rate*. A related term that will be used is the *generation rate*, which denoted the highest rate at which the system generates events. A monitor is required to have a capture rate which at least supports the generation rate of the monitored system, or else there is a risk that the monitor misses events. This makes it vital to determine that the monitor's capture rate can capture all events of the system, i.e., the speeds at which the monitor and the system runs need to be known.

In the case of the SafetyChip its capture rate depends on two properties, the chip's hardware and the schema it polices. The schema and operation of the SafetyChip is given below, in Section 3.

The generation rate of the system can be decided by analysis of the application and real-time kernel (RTK). Since the SafetyChip uses parallelism to mirror and handle multiple tasks of the application the analysis can be performed for each task. The bottleneck that the monitor can be in a multiprocessor system is also eliminated by this parallelism.

An early step in the development of an SafetyChip is formal verification of the run-time application and the RTK. Timing skeletons are created for the application's tasks by means of source code analysis. A timing skeleton consists of a graph describing the timing of a task's interaction with the RTK. Before verification of application properties the skeleton is transformed into the notation of a verification tool, a backend model. The backend model shows more of the information required to create a policing schema so backend models will be used instead of timing skeletons in this paper.

An example backend model for a task  $T_1$  is shown in Figure 3. Task  $T_1$  calls a protected object, an object for mutual exclusion managed by the RTK, and then delays until the protected object should be called again. The backend verification tool is UPPAAL [7] which uses the timed automata

shown in the figure. The task setup used in this paper is the same as that presented in [8] only a slight modification of where the delay is made, to make the task's execution times easier to reason about, has been made. We look at task  $T_1$  to exemplify backend automata. The tasks source code is shown in Table 2.

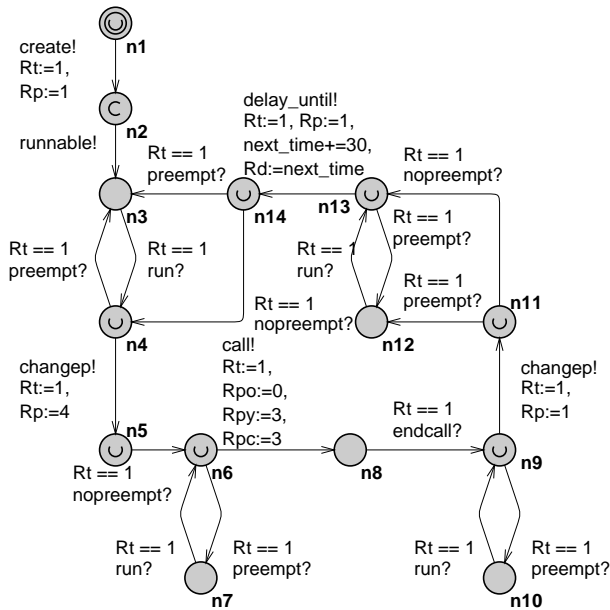
```

1  task T1;
2  task body T1 is
3    T_1 : Task_ID := 1;
4    Next_Time : Time := Start_Time
5                      + To_Time_Span(3.0);
6  begin
7    loop
8      PO.P(T_1);
9      Next_Time := Next_Time + To_Time_Span(3.0);
10     delay until Next_Time;
11   end loop;
12 end T1;

```

**Figure 2: Source code for a task which calls a protected procedure in a timely manner.**

The task calls the protected procedure `PO.P` and then delays for three seconds before calling again.



**Figure 3: Backend model, a timed automaton, for task  $T_1$ .**

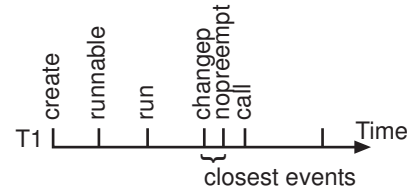
Transitions can have guards, a synchronisation and assignments. For a transition to trigger and transfer the control from one location to another, its guards must be satisfied and in the case of synchronisation another automaton must be ready. Variables (and clocks) are updated according to the assignments when a transition is triggered.

Since the RTK is the component of the system that actively enforces the timing behaviour that the other system components request, the SafetyChip monitors interaction with the RTK. As will be seen, using all RTK events for observation will in most cases be overly detailed, some of the events can safely be left out of the set of observed events.

The top left double-circled location  $n_1$  in Figure 3, with an outgoing transition synchronising on the channel `create`<sup>1</sup> and assigning values to the variables `Rt` and `Rp`, is the initial location. `Rt` is used for relaying task identities and `Rp` is used to relay priorities to the automaton synchronised with. Since there is no guard on the transition it may be triggered unconditionally. A location with a capital `C` in it is a committed location and transitions containing a capital `U` are urgent. Committed locations are used to make consecutive transitions committed, atomic, and urgent locations are used to prioritise the selection of transitions leaving the urgent location, making them urgent, over time transitions. Time transitions, none visible in the figure since the model only uses them in the RTK, can be taken whenever there are no committed or urgent transitions.

Execution time analysis of the tasks can be used to locate the pair of closest events, as shown in Figure 4. The figure shows one possible branch of timing behaviour that task  $T_1$  can exhibit and all possible timing branches must be examined to find the closest events, i.e., establishing the capture rate required to monitor the  $T_1$ , which can be accomplished using the formal verification.

A low capture rate is desired since it is cheaper and easier to implement. Deeper analysis of the locality of events and timing behaviour of the monitor can be used to improve the capture rate, i.e., help keeping it down.



**Figure 4: A sample sequence of events for task  $T_1$ . The closest events (one pair of them) is marked.**

The closest events in Figure 4 are the pair `change` (for changing the priority of a task) and `nopreempt` (indicating that the task was not preempted). There is however no branching possible after the observation of `change` so the monitoring will use minimal time to continue on to its next state, allowing a high capture rate. The closest events can be detected in the formal verification of the application and system or by analysis of the timing skeletons or backend models.

The discussion on the set of events used for observation stated that some events could be left out of the observed set. The above detected closest event illustrate a pair of events that though satisfying the criteria for closest events is not a very interesting pair. Since there is no branching possible after the `change` event, the time between the two events will always be the response time of the RTK. Using these closest events with a fast RTK is likely to require unnecessarily high performance of the monitoring and hence omission of the `nopreempt` event from the set of observed events should be considered. The SafetyChip observes the events shown in Table 1.

<sup>1</sup>The exclamation mark after the channel name indicates that the channel is used for sending. The corresponding receiving channel has a question mark.

**Table 1: The set of observed events.**

|   |  |
|---|--|
| <code>runnable(<math>T_{id}</math>)</code>            | Task $T_{id}$ is added to the ready queue.   |
| <code>run(<math>T_{id}</math>)</code>                 | Task $T_{id}$ is running.  |
| <code>preempt(<math>T_{id}</math>)</code>             | Task $T_{id}$ is preempted (not running).  |
| <code>nopreempt(<math>T_{id}</math>)</code>           | Task $T_{id}$ is not preempted (still running).  |
| <code>change(<math>T_{id}, T^p</math>)</code>         | The priority of task $T_{id}$ is changed to $T^p$ .  |
| <code>delay_until(<math>T_{id}, T^p, d</math>)</code> | Task $T_{id}$ delays until $d$ at priority $T^p$ .   |
| <code>call(<math>T_{id}, PO_{id}, C, K</math>)</code> | Task $T_{id}$ calls protected object $PO_{id}.C$ which is of kind $K$ . $K$ indicates if $C$ is an entry, a procedure or a function. |
| <code>release(<math>T_{id}, PO_{id}</math>)</code>    | Protected object $PO_{id}$ is released.  |
| <code>barrier(<math>B, v</math>)</code>               | Barrier $B$ is set to $v$ . Barriers are used to guard access to protected objects.  |

Techniques like buffering, that can minimise the effect of local event bursts, can also be used to improve the capture rate. Buffering can also be used to overcome other hardware limitations, e.g., to handle schemas larger than the hardware’s default memory can hold.

### 3. SYSTEM POLICING

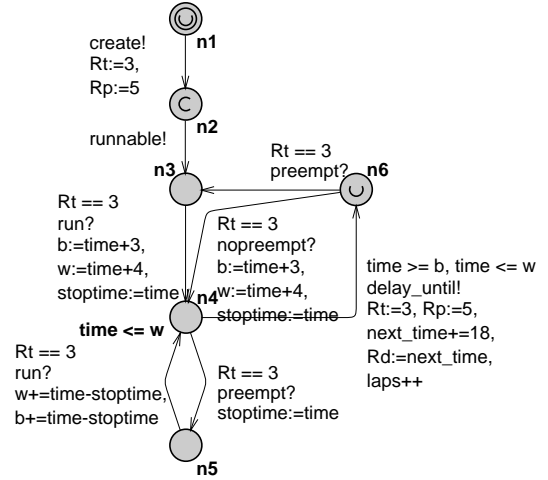
Part of the SafetyChip works like a monitor, but the design also includes an on-chip policing part similar to the external policing device described in [6]. The policing part ensures that the systems operation conforms to a policing schema distilled from the timing skeleton. The policing is accomplished by constant monitoring of the system. If a deviation from the schema is detected, policing is initiated and interrupts are generated to inform the system of this. An on-chip policing device furthers the advantages gained from an on-chip monitor, e.g., the interval between the detection of an error to the policing has been initiated can be shortened considerably.

To illustrate a sample schema we will look at the backend model for a task,  $T_3$ , which executes with a cycle time of 18 is shown in Figure 5. Location  $n_4$  describes a state where the task executes a block of code which has the best execution time (BCET) of 3 and worst execution time (WCET) of 4. The location has an invariant attached to it which is used to force the automaton to leave the location when the worst execution time has been reached, i.e., when the invariant is no longer satisfied.

The schema for the task  $T_3$  should police that

1. the task is made ready to run, `runnable`, every  $18^{th}$  time unit,
2. and that the BCET and WCET of location  $n_4$  are observed.

It is obviously important that the policed schema contains properties that can be guaranteed. For example, the first property states that the task *is made ready to run*, inserted into the ready queue, not that it *is started*, loaded in a processor, in a repeated manner. The difference being that there is no guarantee that a task will start when it is made ready, there can be tasks of higher priority delaying the actual start. That a task is made ready to run can be observed by looking for the `runnable` event (with the correct task parameter relayed), but this event is only generated by the  $T_3$  in its initial behaviour. Once it has executed its code block once it will make a delay and with this surrender control to the RTK’s delay queue. The delay queue will generate



**Figure 5: Backend model, a timed automaton, for task  $T_3$ .**

subsequent runnable events, i.e., knowledge of the RTK is required to distill the schema even when the detailed backend model is available.

#### 3.1 Monitor Automata

The SafetyChip uses, as mentioned above, parallelism to reduce the bottleneck it might otherwise introduce into final system by means of low capture rate. The parallelism is implemented using one hardware automaton, called monitor, for each policed task. The reduced analysis need and the higher capture rate, c.f. Section 2.1, compensates the cost of each monitor being a separate hardware automaton.

Each schema contains an initial schema location for the monitor and then describes transitions to other locations depending on timing and events. Timing is described as the earliest and latest time the observation of a specific event is correct. If the event is observed before the earliest expected time a deviation from the schema will be indicated. A deviation is also indicated if no event has been observed before the latest time of any event observed in a state.

The operation of a monitor can be summarised as follows: When a monitor observes an event on a transition leading from its current location, it must decide the appropriate action. If the event is observed within the correct time, the monitor should proceed to the corresponding location and

continue its monitoring there whereas a deviance should be indicated if the event is observed out of time. The time to find the next state can be calculated by counting the number of states that can be reached from the current state and the worst time needed to transfer control to a new location. Hence the capture rate of the monitor can handle can be decided by a straight forward analysis of the hardware synthesised for it. The capture rate of the chip will be that of the slowest monitor.

The automaton monitoring task  $T_3$  is constructed to reflect the backend model shown in Figure 5. A schema for the task is shown in Figure 2.

**Table 2: Schema for task  $T_3$ . The location names from Figure 5 are shown inside parentheses for node rows.**

| Address | Branch | Call / Node    | BCET | WCET |
|---------|--------|----------------|------|------|
| 00      | 00     | node ( $n_1$ ) | 0    | 0    |
| 01      | 02     | runnable       | 0    | 0    |
| 02      | 03     | period         | 18   | 18   |
| 03      | 00     | node ( $n_3$ ) | 0    | 0    |
| 04      | 05     | run            | 0    | 0    |
| 05      | 00     | node ( $n_4$ ) | 3    | 4    |
| 06      | 07     | delay_until    | 3    | 4    |
| 07      | 00     | node ( $n_6$ ) | 0    | 0    |
| 08      | 03     | preempt        | 0    | 0    |
| 09      | 10     | nopreempt      | 0    | 0    |
| 10      | 05     | period         | 18   | 18   |

The memory address, first column of the schema, is used when addressing the next position of the automata. The second column, branch, is set to the row the monitor should continue from. The branch value for node rows are 00 indicating that the monitor should wait for an event. The Call/Node column holds the node name or the event which should be observed or 'node' if the row is a node row. The BCET and WCET columns hold the times between which the observation of the event follows specification. For nodes the execution time columns hold the best and worst for all events waited for. If the waiting time passes the node's WCET the execution does not follow the schema. There is an additional column, not shown, which can be used to distinguish otherwise ambiguous calls, e.g., calls to different protected objects.

The monitor automata also monitors run and preempt events, even if not present in the schema. These events are, together with information on the current row, used to decide when the execution time counters the monitors keep should be running. A run event indicates that the execution time should be counted if the current row has a BCET and WCET.

The SafetyChip framework also supports the usage of special events that can be inserted in the schema to inform the monitor that it should monitor or check something extraordinary. One such extraordinary check is that of periods. The period event indicates that a new event instruction must be observed before the time indicated by the instruction runs out. This behaviour is implemented as a subcomponent of the monitoring automata and lines with this type of events are inserted at the beginning of each new state. Rows containing this kind of events are processed immedi-

ately when monitoring reaches them. Note that all rows in the schema negatively affect the capture rate that can be supported. And that each added functionality leads to that additional resources will be used for the implementation of the monitor.

A mechanism for adapting and configuring set of monitored events for each monitor automaton has been tested. This mechanism helps the monitors avoid having to observe all events and can increase the chip's capture rate.

## 4. IMPLEMENTATION

A prototype SafetyChip has been implemented in VHDL and synthesised for FPGA. The same size against speed tradeoffs made when implementing the kernel [9], must be made when implementing the monitoring and policing. The two main strategies for the implementation was one automaton for each monitored entity or one automaton handling all the entities. The choice of which of the two strategies to use will determine the tradeoff between the resources available to the system. The implementation described here is optimised for speed rather than for a small footprint. Using the chip are to support high capture rates does however this limits the number of tasks that can be supported.

When an event is observed the monitoring automaton uses 2 kernel clock cycles per transition check (row in Table 2). If there are 2 possible transitions for an automaton, like from location  $n_6$  in the example schema, and the last of the listed ones is observed, updating the automaton uses  $2 * 2$  clock cycles. Thus the fastest operational capture rate can be decided by locating the largest number of events observable from the same state and using this as the worst case.

An obvious optimisation is to order the transitions in the order of likelihood so that the most likely is the first to be considered. This has not been implemented but would use the same kind of reasoning done in branch predicting compilers.

If the size of the collected schemas or the number of automatons required for the monitoring exceeds the capacity of the hardware it is possible to use (a hierarchy of) caches to place the schemas off chip. This has, however, not been implemented.

An improvement to the SafetyChip lies in extending the monitored set of events. The current chip monitors assertion events in addition to the events concerning the real-time behaviour of the monitored system. The use of the assertion checking is semi-intrusive as it in some cases require extra instructions to be inserted into the system. These instructions can however be included in the verified model, allowing the effects of them to be verified and included in the monitored behaviour.

### 4.1 Implementation Results

The implementation was synthesised for a Xilinx Virtex-II Pro 2VP30.

A system consisting of monitors for 32 task and supporting 32 bit time uses less than 40% of the programmable resources and around 70% of the on-chip memory blocks and it could be run at frequency of 150 MHz. This system can allow 75000 branching alternatives in the schemas and still support a capture rate of 1ms.

The size used by each event row used for the monitoring (Table 2) requires around 90 bits. The size used depends on the number of bits needed to describe the branching,

the calls, the execution times. There is a 18 kb memory (BRAM) available to each monitor which makes it possible for it to fit 200 rows.

## 5. CONCLUSION AND FUTURE WORK

The SafetyChip (SC) proposes a way to reuse the investments made in formal methods during the life time of a system. It also proposes the use of automated model extraction which enables formal verification to be introduced into development with minimal investment in the creation of models, a traditionally expensive process.

The current trends in microprocessor development points towards adding FPGA to all microprocessors. Current developments in FPGA extends embedded microprocessors with monitors for off-chip analysis of the run time behaviour of the FPGA system. A possible next step is to add the policing features of the SC to further utilise the possibilities of the FPGA.

The SC architecture uses the extracted model to non-intrusively monitor the running system and, if the system expresses behaviour outside the modelled behaviour, the SC can take control and police the systems execution, e.g., by generating interrupts.

## 6. REFERENCES

- [1] A. Burns, B. Dobbing, and G. Romanski, "The Ravenscar Tasking Profile for High Integrity Real-Time Programs", *Reliable Software Technologies — Ada-Europe 1998*, LNCS 1411, Springer-Verlag, 1998.
- [2] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the Ada Ravenscar Profile in high integrity systems", *University of York Technical Report YCS-2003-348*, 2003.
- [3] S. Evangelista, C. Kaiser, J-F. Pradat-Peyre and P. Rousseau, "Verifying Linear Time Temporal Logic Properties of Concurrent Ada Programs with Quasar", ACM SIGAda Annual International Conference—SIGAda 2003, Ada Letters XXIV(1), ACM, 2004.
- [4] T. Gerdsmeyer and R. Cardell-Oliver, "A Method for Verifying Real-Time Properties of Ada Programs", Proceedings of the 7<sup>th</sup> IEEE International Conference on Engineering Complex Systems (ICECCS 2001), IEEE, 2001
- [5] Altera Corporation, SignalTap II.  
<http://www.altera.com/>
- [6] ARM Limited, "Using EmbeddedICE", Application Note 31, 1999.
- [7] K. Larsen, P. Pettersson, and W. Yi, "Uppaal in a Nutshell", International Journal on Software Tools for Technology Transfer, Springer-Verlag, 1997.
- [8] K. Lundqvist, and L. Asplund, "A Ravenscar-Compliant run-time kernel for safety critical systems", Real-Time Systems, 24(1), 2003.
- [9] G. Naeser and J. Furunäs, "Evaluation of Delay Queues for a Ravenscar Hardware Kernel", MRTC report ISSN 1404-3041 ISRN MDH-MRTC-176/2005-1-SE, Mälardalen Real-Time Research Centre, 2005.
- [10] Xilinx Inc., Chipscope integrated logic analyzer, San Jose, CA 95124-3400, 2000.  
<http://www.xilinx.com/products/chipscope/>
- [11] R. York and J. Sharp, "Real-time debug for systems-onchip devices", ARM Limited, June 1999.