Temporal Skeletons for Verifying Time

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

Kristina Lundqvist
Dept. of Aeronautics and Astronautics
Massachusetts Institute of Technology, USA
kristina@mit.edu

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

ABSTRACT

This paper presents an intermediate notation used in a framework for verification of real-time properties. The framework aims at overcoming the need for the framework user to have significant knowledge of the verification specific detail that formal verification at some level is bound to impose on a model. In order to accomplish this, model extraction from source code of an initial formal model, a timing skeleton, is made automatically.

The model refinement needed to transform the temporal skeleton into a model that can be verified is not done immediately. This allows postponement of the abstraction and specialisation needed for the verification which further improves the readability of the skeleton. The purpose of the timing skeleton is that it easily can be validated to represent the source code it was created from. The timing skeleton is then automatically refined with verification detail, and then hidden for the user, transformed into the notation of a verification tool. This transformation is hidden from the user.

In order to reduce the complexity of the application model the framework uses a formally verified run-time kernel with a clear separation from the application. The kernel supports preemption, dynamic priorities and multiple processors.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal Methods

General Terms

Verification

Keywords

Formal Notation, Verification, Ada

1. INTRODUCTION

The number of high-integrity, or safety-critical, systems created is increasing not only with new applications but also with the rewriting of old ones. Both the new and old applications are likely to face the question of technology choice. This paper looks at a framework where tasking, traditionally not among the choices, can be added to the list of available technologies for use in high-integrity systems.

Tasking systems with preemption have not been widely introduced into the area of high-integrity systems since it generally is hard to determine the real-time behaviour of tasking systems. The most widespread execution model for safety critical high-integrity systems today is that of cyclic executives without preemption. The advantages of cyclic executives systems like that is that an execution schedule for the system can be created beforehand and that the real-time behaviour, following the schedule, is well defined.

The run-time behaviour in the alternative execution model, with tasking and preemption, is not as easy to decide. Allowing preemption can make better use of the resources available in a system as tasks are not bound to specific execution slots. However, an obstacle with the introduction of tasking is that the complexity of the system's timing behaviour increases. The high level of analysis demanded by safety critical systems may therefore seem to prevent the use of tasking. An effort to allow reduced tasking in high-integrity systems is collected in the Ravenscar tasking profile [4, 5]. Ravenscar is a subset of Ada and has been developed to reduce the nondeterministic behaviour of tasking by making restrictions of language features like dynamic creation and termination of tasks, and task communication. In order to allow tasking systems to be used in high-integrity application areas the behaviour of the system must be well defined and analysis must be possible and the Ravenscar profile restricts the tasking of Ada to impose near determinism.

The SafetyChip project, with teams at Mälardalens University and MIT, is developing a monitoring and policing device for safety critical embedded systems. The project consists of several packages, including source code transformation, formal verification and hardware synthesis. Two real-time kernels (RTK) for the Ravenscar profile have been developed [23, 27]. Both kernels are designed to be customised and application tailored, then synthesised to hardware. The kernels have been formally modelled and verified together with sample applications to assert that they exhibit the demanded properties. The kernels themselves are not verified stand alone with parameters set to work with a number of application. Instead verification is done of the complete system, with both the kernel and an application, ensuring that the system conforms to the specified behaviour, not that the kernel or application alone does.

SIGAda'05, November 13–17, 2005, Atlanta, Georgia, USA.

The introduction of formal modelling of applications is easiest at early stages of the development. At the initial stage the application is most likely to be described in abstract models without all the detail, noise [7], that the programmers will have to add to make the application work. Detail will both make it harder to create the model, *i.e.*, choosing what detail to include, and it will obscure the model if included when it should not have been. Keeping the model up to date with the application as it is developed and maintained is similar to the problem of specification management.

Even as the development of an application may benefit from the use of formal notation and methods early in its development [25], demanding that the development process conforms to one specific process would greatly limit the usage of a verification framework. Hence the framework presented in this paper does not make any assumptions about how a system has been created, it works with the source code of the actual application. Late introduction of a framework for formal methods into the development of an application relies on the nontrivial ability to abstract the application or parts of it, *i.e.*, removing noise.

Limitations of the framework are that it requires a model of the kernel the final system uses, and that it requires the availability of execution times for code segments. The first limitation is eliminated by the use of a custom kernel, and an assumption of the availability of execution times is used to handle the second.

One challenge in verifying high-integrity systems, as well as making verification more widely used, lies in making the formal models usable and understandable to the developers that create the system and to the technicians maintaining it [17, 19]. In this paper we present a framework addressing these issue by having the run-time system and the application separated, as suggested in [15], and by using a source code based modelling of the application's temporal behaviour. The model construction process consists of the five stages shown in Figure 1;

1. Ada source code is automatically transformed into
2. a temporal skeleton, a model in an intermediate readable notation, which is further transformed into
3. a backend model. The backend model is together with
4. a backend model of an RTK combined into
5. a verification model which can be verified using a tool, *e.g.*, UPPAAL [2, 3, 18].

This paper focuses on the transformation from the source code into the notation of temporal skeletons.

The RTK is a generic component kernel, not an integrated part of the application, mainly for two reasons *a*) it can be reused between designs and *b*) its component interface makes it easy replaceable with another kernel. The actual kernel is implemented in hardware, making its run-time behaviour more deterministic. The alternative, especially when working with a hardware software co-design, is integrating the RTK into the application itself. However, an integrated kernel increases the complexity of the application and is likely to make the whole design much harder to model and understand, *i.e.*, the level of noise is unnecessarily increased. Keeping the complexity of the application down allows it to be expressed using a notation closer to its source, making it easier to validate that the model models the application.

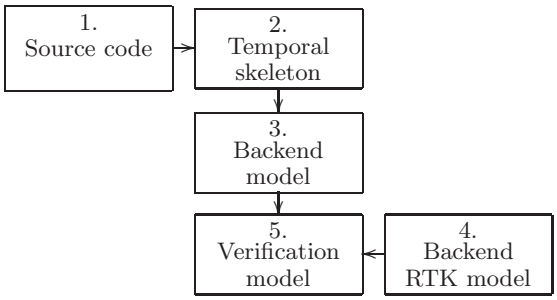

Figure 1: A framework for formal verification of source code.

Different approaches have been used to verify temporal behaviour of Ada programs. In [9] HyTech is to verify run-time behaviour of conservative abstractions of general Ada. Models are created in two steps; the first step transforms the program into a transition system and a second step further transforms it into the hybrid automata used in the verification. In [14] Ada is transformed into TLA+ which is abstracted to what the UPPAAL tool can interpret. The ability to compute the execution times of sequential code is assumed and verification of real-time properties is then made of the application together with reduced RTK functionality. In [6] annotations with temporal information are added to the source code and then used when verifying the system's timing behaviour. In [16] extensions to the SPARK framework, for verification of sequential code, are proposed. PSV is used to verify a state transition system based on the extensions where the transitions represent either sequential code or calls to protected objects. In [13] the Quasar tool [12] is used to verify concurrent programs. Quasar uses ASIS, the Ada Semantic Interface Specification, to create a model used in the verification.

Works like [22, 24], where it is shown how verification can be used to gain knowledge of real-time behaviour and execution times of systems, also motivate the use of verification in real-time systems.

2. TIMING SKELETONS

One of the main aims of the temporal skeleton model is that it should be readable by the non-expert developers of the system, not only by verification experts. This makes it necessary to structure the intermediate model in a way identifiable according to blocks describing some functionality of the source code. Graphical approaches for making formal methods, especially model-checking, understandable to non-specialists have been presented, *e.g.*, in [11, 21, 26, 29].

We have chosen to focus our notation of time on the application's source code for the main reason of making validation easier. Validation is carried out to ensure that the model captures the behaviour of the application and this is easier if the model in itself resembles the application's source. A notation with limited resemblance to the source code obviously requires more domain knowledge of the party validating the model.

The model of a system benefits from a clear definition of the system components and the properties of the compo-

nents. All timing behaviour of an application is in some way managed by the RTK, but it is the application tasks that create the temporal behaviour by requesting services from the RTK. Tasks themselves either directly ask the kernel for a specific action, for example by requesting to be delayed, or are controlled by the kernel in response to the requests of other tasks, *e.g.*, when the kernel preempts a running task to run another. We use the communication between the kernel and the application as a base for describing the timing behaviour of the application.

The actions of a running task, *i.e.*, instructions executed by the task, can be categorised as either active or passive according to whether or not execution of the instruction will request changes in the temporal behaviour from the kernel.

Definition A: Active instructions. *Instructions that either actively interact with the RTK to or in other ways alter execution of the system are defined as active. The resulting alteration can change the execution immediately or with delayed effect.*

Instructions controlling time directly, by asking the kernel for a specific service like the `delay until` which instructs the kernel to delay the task, are defined as active. The definition of active instructions defines a set of instructions that depend on the current abstraction of the system. Normally the set of active instructions corresponds to the interface calls to the RTK but instances of some instructions, like assignment, can in certain models be considered active, *e.g.*, when they change the premises for the effects of active instructions. The complement to active instructions are called *passive instructions*.

Definition B: Passive instructions. *Instructions whose only effect on the timing behaviour of the system is their consumption of processing capacity are defined as passive.*

Passive instructions still play a role in the execution of the system as a task executing a block of them can be preempted by the effects of the active instructions of another task. The two sets of instructions are disjunct even if the consumption of processing capacity is not limited to the passive instructions. An important property of passive instructions is that they have best and worst case execution times, BCET and WCET, which reflect their execution times in the running system. The effects on the system of active instructions can be separated into a computational part and a part interacting with the kernel. The computational part can can be treated as passive and modelled to embody any time used by the active part leaving the active part consuming no time [23]. For example, an instruction delaying until a specific time will first compute the time which it should delay to and preparing the kernel request in a passive part, and then call kernel functionality with that time in an active part. Time used by the active part can easily be attributed to the passive part without disrupting the timing behaviour of the model.

To structure the models we define *control flow*, *passive blocks* and *timing skeletons*. Blocks will be used to collect sequential instrucitons into larger chunks and skeletons will add ordering and control flow among the blocks.

Definition C: Control flow. *Instructions have a set of next instructions. The flow to each next instruction can be attributed with a guard stating the conditions when the respective next instructions can be reached. The guards range over the variables available to the instruction the flow leaves. Normally there is only one instruction in the set of next instructions that have a guard which allows it to be reached, i.e., flow is deterministic.*

Control flow adds the ability of choice to the modelling and allows us to create structures for conditional execution and different kinds of loops. Next, to improve readability, and make the model better suited for verification, we define a way to merge instructions into larger entities.

Definition D: Passive blocks. *A passive block is constructed from a sequence of successive next instructions. There are two rules to what instruction can be included into a block 1) the first instruction in the block is the only block instruction that is the next instruction of any instructions outside the block and 2) the last instruction of the block is the only block instruction that has as next instruction an instruction not inside the block. Also, 3) each instruction may occur only once in a passive block.*

Passive instructions can be turned into passive blocks and then merged with other, through control-flow connected, passive blocks as long as these properties are sustained. The need to restrict the number of times an instruction may occur in a block is due to the fact that infinite loops otherwise could lead to blocks with an infinite sequence of instructions. With the restriction the loop itself will describe a passive block. It is possible that a passive block contains subroutine calls, *i.e.*, procedure and function calls, as these can also be categorised as active or passive according to whether or not the subroutine contains active instructions.

The next definition we make is that of *timing skeletons* which glue passive blocks with active instructions.

Definition E: Timing skeletons. *Sequential active instructions and passive blocks are collected into timing skeletons. A timing skeleton has at most one initial instruction and may or may not have a final instruction. Each timing skeleton has a kind denoting the kind of the entity the skeleton describes, e.g., task or subroutine.*

With all the pieces we need to describe the timing behaviour of the different components of a system in place we can now describe a system as a collection of temporal skeletons. To make our modelling language even more useful we allow instructions to invoke skeletons, *i.e.*, the skeletons can be used to describe subroutines which can be called by other skeletons. If a timing skeleton contains no active instructions it can be *inlined* in the passive blocks of a caller. For example, a subroutine call to a subroutine which is described by a passive block constitutes a passive instruction and can as such be merged with passive instructions surrounding it. Inlining enables passive blocks to include large spans of sequential instructions into single passive blocks which keeps the total size of the final timing skeleton down.

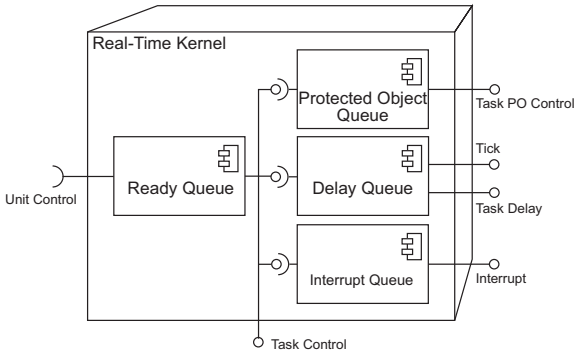

Figure 2: The Kernel components.

3. AUTOMATED MODELLING

A tool using the definitions given in Section 2 was implemented using ASIS [1] to explore how the temporal skeletons would perform for sample code. The ASIS framework can be used to traverse code, *e.g.*, as in [8] to validate if it conforms to Ravenscar profile. ASIS allows analysis to span multiple compilation units, like packages, which creates an environment where a project can be analysed without any need for initial grooming of its source code.

In our approach the units are traversed and an initial skeleton is created. Skeletons are created for the tasks of the application and, in the Ada case, for procedures and entries of protected objects. The initial skeletons outline the control flow of the application and each instruction constitutes a skeleton location of its own. Locations containing active instructions are marked as temporal and passive blocks are created by merging non-active locations, as outlined in Section 2. Extra detail, *e.g.*, about execution times and control flow information, like loop bounds, can be added to the model at this state to show this to the user, or it can be added later, in the model specialised for the verification tool.

The skeletons can at this stage be inspected to validate their conformance to the source code. In many cases the skeletons can be superimposed over the original source code to make the source code conformance easier to validate.

3.1 The Run-Time Kernel

The clear separation of the instructions into those controlling the kernel, active instructions, and those controlled by the kernel makes it natural to model the application and kernel separately. There are several benefits from this, *a*) the size and complexity of the model application are not affected by the kernels ditto, *b*) it is easy to change the kernel without having to remodel the application, and *c*) it makes it easy to reuse the kernel and its model. The kernel components are shown in Figure 2.

To make the application skeletons small we have separated the run-time kernel functionality from the application. All run-time functionality has been moved to an RTK model which is merged with the application prior to verification. The RTK consists of kernel components for a ready queue, a delay queue, a manager of protected objects, and a component for interrupts. There are also models of hardware like processing units and clocks the kernel interacts with. Calls to the external interfaces of the kernel components, *i.e.*, the interfaces that the application can use to interact with the kernel, are not introduced in the timing skeletons. The skeletons are concerned with notation only and do not depend on the use of any specific kernel. The connection to a kernel is added once the timing skeletons are transformed into the notation used for verification, where the kernel is need to supply the timing policies.

3.2 Protected Objects

A mechanism for mutual exclusion found in Ada is the protected objects [28]. The objects are interesting since they are a source of non-determinism in the Ravenscar tasking profile, as will be explained below.

Protected objects basically create mutual exclusive access to code and variables much in the same way as semaphores do. A protected object has a data part which contains the protected variables and a code part which contains subroutines for accessing the variables. Subroutines can be functions, procedures and entries. Protected functions can be executed concurrently, *i.e.*, they do not enforce mutually exclusive execution, but may, to compensate this, only read protected variables, not change them. Protected procedures and entries may only be accessed mutually exclusively and may change variables protected by the object. A protected entry has a barrier which guards if the entry can be called or not. If the barrier of an entry evaluates to false, *i.e.*, it does not allow the calling task to enter the entry and execute its code and the task will be suspended until the barrier allows entry. When a task is suspended the task will be in a queue for the entry. In multiprocessor systems tasks can also be queued for access to protected procedures[1]

The number of entries of a protected object is in the Ravenscar profile limited to one using the restriction

```
Max_Protected_Entries => 1
```

and in the same way the maximum length of the queue of suspended tasks is limited to one suspended task

```
Max_Entry_Queue_Depth => 1
```

All restrictions made in the Ravenscar profile but for the dynamic restriction limiting the number of tasks queued waiting for access to an entry, can be checked with static analysis of source code. In order to verify if an application preserves the dynamic restriction, verification needs to be done using the execution times of the application or by other means of analysis which takes into consideration the systems run-time behaviour.

Access to a protected object's procedures and entries is active in its nature since it, when the call is made, is not generally known whether or not the calling task will be suspended. Hence we include the protected objects as an integral part of our model and kernel.

4. AN EXAMPLE APPLICATION

In this section the skeleton of an example task will be presented to illustrate what they look like. The example will use the source code of a task as input. The task `Task2`, depicted in Figure 3, repeatedly opens a protected object

[1]No queueing occurs on procedures in single processor systems as a result of a the priority ceiling policy and tasking of Ada. However, the mechanisms do not ensure the same for systems with multiple processors.

entry using the protected procedure `PO.Open`. If the global variable `Global` is 10 the task will call the entry `PO.Call` and otherwise it will go back and open the object.

```
1  task Task2;
2  task body Task2 is
3     T_2 :  Task_ID := 2;
4  begin
5     loop
6        PO.Open(T_2);
7        if (Global = 10) then
8           Global := 0;
9           Work(0.2,0.3, T_2);
10          PO.Call(T_2);
11       end if;
12    end loop;
13 end Task2;
```

Figure 3: The source code of the task `Task2`. The task competes with other tasks in calling `PO.Open` and it it is possible, but uncertain, that it will call `PO.Call`.

The instructions are identified as active or passive. Passive instructions are turned into passive blocks which are merged to form larger blocks. The result is shown in Figure 4.

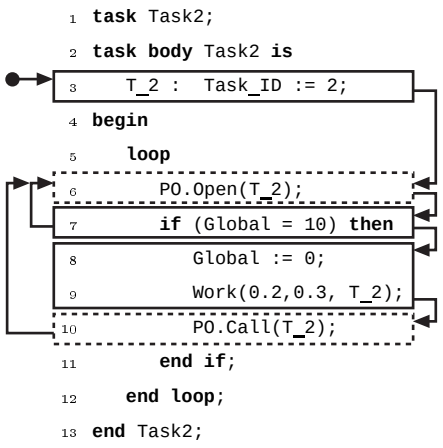

Figure 4: An automaton for the task `Task2` superimposed over the task's source code. There is a total of five locations, two representing active instructions (dashed line) and three representing passive blocks (solid line). The dot-and-arrow indicates the first instruction (block) of the task.

Sequential passive instructions do not in themselves add locations to the skeleton. This favours large sequences of passive instructions as they will contribute little in making the complexity of the skeleton grow. Active instructions, however, will increase the skeletons size which is easily motivated by the fact that it is the timing behaviour that the skeletons are supposed to describe.

There are a few things that can be noted in the time skeleton. First, the conditional execution has not been automatically transfered to the skeleton, *i.e.*, the value of the variable `Global` is not present in the model. This can be added to the skeleton if it is needed for the verification but it is not added to the timing skeleton by default. The loop line in the source, line 5, and the lines closing other constructs, lines 11–13, are not part of any instruction block since they do not add any behaviour to the skeleton that is not already present in the skeletons transitions. The source line with the `if` conditional, line 7, is in a block since it contains a comparison which uses execution time. If the two calls to the protected object, lines 6 and 10 had been to a non active subroutine instead, the whole body of the task, *i.e.*, lines 2–13 could have formed a single passive block. Though the closing lines would, following the reasoning above, not need to be included in the block.

The skeletons can be decorated with flow information, *e.g.*, the value of the `Global` variable can be traced and incorporated in the skeleton using flow analysis. Each passive block consists of sequential code for which a BCET and WCET can be derived at the desired granularity. The derived times can then be used to decorate the skeleton to create a skeleton which can be used for verification of the temporal behaviour of the system.

The skeletons also hide the actual kernel interaction from the user, interaction that is not shown by the application source to start with. Hiding this kind of informational noise from the user allows the user's focus to remain with the task of validating that the skeleton describes the source code. Any added noise is likely to distract the user's attention and should therefore be avoided in the skeleton. The detail will be added to the automaton used in the verification.

5. A BACKEND MODEL

The chosen backend for modelling the run-time kernel and the systems is Uppaal [18]. The main reason behind the choice was that this would make it easier to compare the RTK with the kernel presented in [20], which describes the kernel used in [27]. Uppaal uses timed automata to model, simulate execution and verify systems. The transformation tool was extended with a backend which refines temporal skeletons into Uppaal's timed automat. The transformation uses a set of fixed automata patterns to construct timed automata which are compacted to reduce any unnecessary artifacts introduced by the application of the patterns. The patterns transform the skeleton from stage 2 to stage 3 (according to Figure 1) and a backend model for `Task2` is shown in Figure 5.

The timed automata representing the RTK are added to the backend skeleton to form the verification model whose properties are investigated using the verification tool.

The RTK automaton for handling protected objects contains a location, $PO.n6$, which indicates that a task has tried to queue for an entry for which there is another task queued. Verification of the dynamic Ravenscar restriction of the entry queue length in a system can in Uppaal be made using the query $\exists\Diamond(PO.n6)$, *i.e.*, it is investigated if the error location eventually can be reached in any execution of the system.

Other temporal properties that can be verified are, *e.g.*,

- if a specific passive block can be, or always will be, reached before or after a given system time

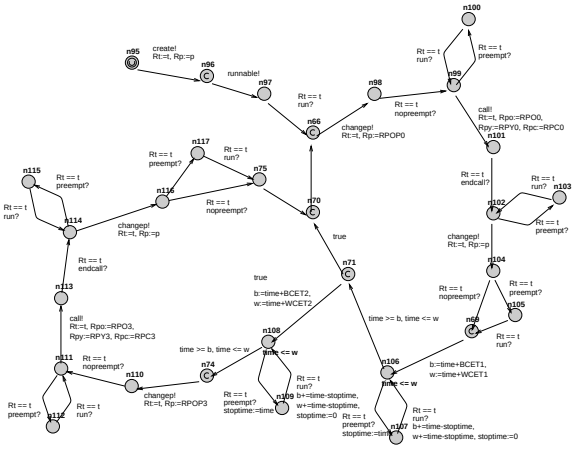

Figure 5: The automaton used in the verification of Task2. The complexity of the automaton has increased substantially compared to the automaton found in Figure 4.

- how many times a specific task will be preempted
- how many times a loop can be taken, *etc.*

In general, the tool allows verification of *a*) reachability and deadlocks, and *b*) global properties, like that certain conditions never occur during execution. Transitions can be attributed with counters to gain detailed information about possible executions, *e.g.*, the number of times a specific transition can or will be be used.

6. EXPERIMENT - ARGUS ROBOTS

The real case for the tool chain has been software that has been developed at both Uppsala University and Mälardalen University. The application is the control software for soccer playing robots. The robots have been built in several consecutive student projects, and have involved student from several different curricula [10] ranging from mechanical engineering over electrical engineering and computer engineering to computer science.

6.1 Experiment Description

The robots are used in the robocup competition[2] in the middle sized league. The requirements on these robots are that no global sensors are allowed, and the size is 50 cm in diameter and at least 40 cm high but less than 80 cm. The weight of our robots is limited to 18 kg. The robots have omnidirectional drive, *i.e.*, a robot can travel in any direction and rotate around any point inside or outside the robot's diameter. The sensor system include two or more video cameras, ultra sonic range sensors, infrared range sensors, up to 24 reflective sensors under the robot, and sensors that indicate that the ball is in the *kick postion*. Communication between robots and a server that is off-field is by VLAN (802.11b). All sensors are handled by a microcontroller (Atmel Mega16) and the communication between these controllers and the main CPU is by a CAN-bus. Later robot versions use an optical CAN bus which allows a more modular robot design.

[2] `www.robocup.org`

All communication is interrupt driven and one task handles all interrupts from the CAN-bus (through a protected object) and messages are directed towards various buffers (implemented as protected objects). Each sensor has its own handler which reads sensor data from the buffer holding data for it. The entry of the protected object is used for this access. A similar setup is used for the VLAN-communication. The is a sensor-fusion layer which builds a world-view, that both contains the own robot's position and direction as well as other robots' positions (including opponent robots') and the ball's. Tasks in this layer use data from the sensors on the robot as well as world-view data from the other robots. The result of the sensor-fusion layer is sent to the strategy layer, where an AI-system makes decisions about what actions the root should carry out, *i.e.*, where to move, how to rotate and if the kicker mechanism should be fired.

The control software is written in Ada using the Ravenscar profile. The first version of the robots used a standard PC-104 with LINUX as operating system. Subsequent versions use the Xilinx Virtex II Pro chip which has an in-built PowerPC. VxWorks is used to control the PowerPC.

The software has evolved over several years, and the students have a chance to work with real code, and have to face all the issues that are related to maintain *other students* code. The size of the problem makes it impossible to rewrite the whole of it every course instance. It is not only a stimulating project, since there are so many challenges, but it is also a project which resembles a real world system. Still, the system is in-house and there are no hassles added by, *e.g.*, the NDA's which would be required when working with industrial software.

6.2 Experiment Results

Model extraction distilled the 16 tasks and 30 protected objects that the program consisted of. Most tasks and protected objects were simple, but a handful were complex and interacted with the simpler ones using the protected objects.

The first transformation phase created initial skeletons that together used approximately 3000 locations and 1400 transitions. By the end of the passive block phase, where sequential passive instructions had been merged into passive blocks, the model had been reduced to 700 locations and less than 800 transitions. It can be noted that the construction of passive blocks identified one passive block consisting of 140 lines of source code and that this block called many routines in other packages. The model extraction identified and inserted instructions for the updating of the 20 barrier variables used by the protected objects.

Most of the protected objects are simple and used to block or unblock the protected object's entry but they all have unique identities and values like priorities that need to be set.

The subsequent transformation into the timed automata used by UPPAAL resulted in a model containing 1300 locations and 1700 transitions. The layout of the target automata is also automated and tries to minimise the number of crossing edges while keeping the locations as close as possible together. The algorithm works fine for small graphs but will result in a coarse layout for larger automata. The automaton for the Start_Player task is shown in Figure 6.

Verification was attempted but, as expected, dead-locked fairly promptly due to the lack of a real-world model needed to generate the startup data the controller task requested

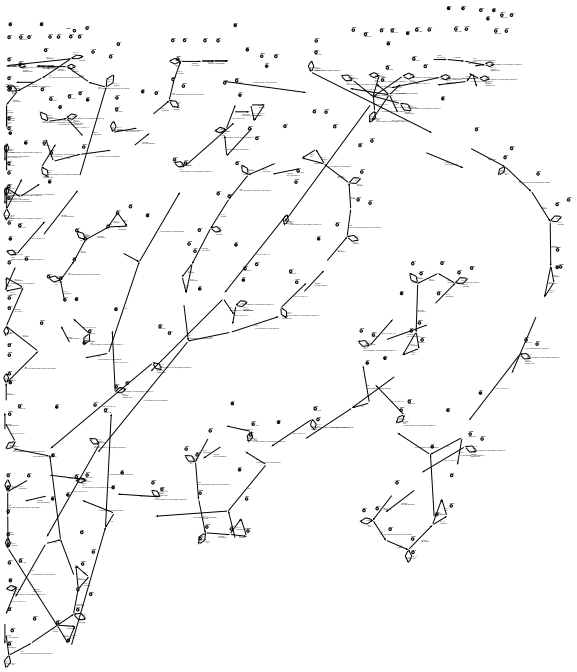

Figure 6: The target automaton for the main task, Start_Player, of the Argus robots.

from external sensors and units. The creation of a model of the environment was not attempted in this experiment.

7. CONCLUSIONS AND FUTURE WORK

This paper presents a notation of timing skeletons that can be used to describe a tasking application. Superimposing the skeletons over the source code is straight forward which makes it easy to validate that they model the source. The low detail of the skeletons, *i.e.*, the intentional hiding of detail not needed prior to the verification, makes them easier to understand, perhaps even understandable to non-specialists in verification. The skeletons can be extended and annotated with additional information to add, *e.g.*, control flow information and guidance to the verification.

The skeletons are extracted from source code using a tool to allow the use of them to

- enter the development process late, and
- reduce the risk that they do not describe the system as the system is updated, they can be re-extracted when the application changes.

The notion of passive blocks, sequences of instructions not interfering with the system's time flow, creates a situation where skeletons are less likely to grow in complexity when sequential computation is added to an application. The complexity of a skeleton grows when timing behaviour is added.

There are some areas that can be further improved to create a more automated framework for verification of real-time systems. Annotation of execution times, today added manually, can be automated. This would greatly improve the usability of the framework.

As the verification impact of the RTK backend model is so large it needs to use all power available to the backend verification tool in order to make verification effective. This has lead us not to transform the RTK into the backend notation using a tool. Investigation of the possibilities of automatic backend model generation could be investigated to allow easier use of additional verification tools.

8. REFERENCES

[1] Association of Computing Machinery (ACM) Special Interest Group on Ada (SIGAda) ASIS Home Page. `http://www.acm.org/sigada/WG/asiswg/`

[2] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal", *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, 2004.

[3] J. Bengtsson, and Wang Yi, "Timed Automata: Semantics, Algorithms and Tools", Lecture Notes on Concurrency and Petri Nets, W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, 2004.

[4] 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, L. Asplund, ed., Springer-Verlag, 1998.

[5] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the Ada Ravenscar Profile in hight integrity systems", *University of York Technical Report YCS-2003-348*, 2003.

[6] A. Burns, and T-M. Lin, "Adding Temporal Annotations and Associated Verification to the Ravenscar Profile", *Reliable Software Technologies — Ada-Europe 2003*, LNCS 2655, J-P. Rosen and A. Strohmeier, eds., Springer-Verlag, 2003.

[7] D. Bustard, and A. Winstanley, "Making Changes to Formal Specifications: Requirements and an Example", *Transactions on Software Engineering*, 20(8), IEEE, 1994.

[8] W. Colket, "Code analysis of saftey-critical and real-time software using ASIS", *Proceedings of the 1999 annual ACM SIGAda international conference on Ada (SIGAda'99)*, ACM, 1999.

[9] J. Corbett, "Timing Analysis of Ada Tasking Programs", *IEEE Transactions on Software Engineering*, vol. 22, No. 7, 1996.

[10] M. Daniels and L. Asplund, "Multi-Level Project Work; a Study in Collaboration", *in Proceedings to IEEE Frontiers in Education conference*, IEEE, 2000

[11] N. Dulac, T. Viguier, N. Leveson, and M-A. Storey, "On the Use of Visualization in Formal Requirements Specification", *International Conference on Requirements Engineering*, 2002.

[12] S. Evangelista, C. Kaiser, J-F. Pradat-Peyre, and P. Rousseau, "Quasar: A New Tool for Concurrent Ada Programs Analysis", *Reliable Software Technologies — Ada-Europe 2003*, LNCS 2655, J-P. Rosen and A. Strohmeier, eds., Springer-Verlag, 2003.

[13] S. Evangelista, C. Kaiser, J-F. Pradat-Peyre, and P. Rousseau, "Verifying Linear Time Temporal Logic Properties of Concurrent Ada Programs with Quasar", Ada Letters, vol. XXIV, No. 1, *ACM*

SIGAda Annual International Conference — SIGAda 2003, ACM, 2004.

[14] T. Gerdsmeier, and R. Cardell-Oliver, "A Method for Verifying Real-Time Properties of Ada Programs", *7th International Conference on Engineering of Complex Computer Systems*, IEEE, 2001.

[15] T. Hoverd, "Are Formal Methods The Answer?", *IEE Colloquium on Requirements Capture and Specification for Critical Systems*, 1989.

[16] D.J. Howe, and S. Michell, "An Approach to Formal Verification of Real Time Concurrent Ada Programs", *12th International Ada Real-Time Workshop*, Ada Letters, Volume XXIII, No. 4, 2003.

[17] X. Jia, "A Pragmatic Approach to Formalizing Object-Oriented Modelling and Development", *The Twenty-First Annual International Computer Software and Applications Conference*, IEEE, 1997.

[18] K. Larsen, P. Pettersson, and W. Yi, "Uppaal in a Nutshell", International Journal on Software Tools for Technology Transfer, Springer–Verlag, 1997.

[19] I. Lee, and O. Sokolsky, "A Graphical Property Specification Language", *Proceedings of 2nd IEEE Workshop on High-Assurance Systems Engineering Workshop*, IEEE, 1997.

[20] K. Lundqvist, and L. Asplund, "A Ravenscar-Compliant run-time kernel for safety critical systems", Real-Time Systems, 24(1), 2003.

[21] T. Mertke, and G. Frey, "Formal Verification of PLC-Programs Generated From Signal Interpreted Petri Nets", *2001 IEEE Systems, Man, and Cybernetics Conference*, 2001.

[22] A. Metzner, "Why Model Checking Can Improve WCET Analysis", *Proceedings of the 16th International Conference on Computer Aided Verification*, LNCS 3114, Springer-Verlag, 2004.

[23] G. Naeser, and K. Lundqvist, "Component-Based Approach to Run-Time Kernel Specification and Verification", *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

[24] D. Naydich, and D. Guaspari, "Timing analysis by model checking", *5th NASA Langley Formal Methods Workshop*, 2000.

[25] S. Paynter, "The benefits of formal specification are not automatic", *IEE Colloquium on Practical Application of Formal Methods*, 1995.

[26] M. Pezzé, and L. Baresi, "Can Graph Grammars Make Formal Methods More Human?", *ICALP 2000 Workshop on Graph Transformation and Visual Modeling Techniques*, 2000.

[27] A. Silbovitz, and K. Lundqvist, "A Hardware Implementation of a Ravenscar-Compliant Run-Time Kernel", *The 22nd Digital Avionics Systems Conference*, IEEE, 2003.

[28] A. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf and S. Michell, "Integrating Object-Oriented Programming and Protected Objects in Ada 95", ACM Transactions on Programming Languages and Systems, 1999

[29] M. Zimmerman, M. Rodriguez, B. Ingram, M. Katahira, M. de Villepin and N. Leveson, "Making Formal Methods Practical", *Proceedings of the 19th Digital Avionics Systems Conferences*, 2000.