A Tool for Automatic Model Extraction of Ada/SPARK Programs

André Carvalho   Nuno Silva   Nelma Moreira   Simão Melo de Sousa

Technical Report Series: DCC-2010-05
Version 1.0 August 2010

Departamento de Ciência de Computadores
&
Laboratório de Inteligência Artificial e Ciência de Computadores

Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL

Tel: 220 402 900   Fax: 220 402 950
`http://www.dcc.fc.up.pt/Pubs/`

A Tool for Automatic Model Extraction of Ada/SPARK Programs

André Carvalho[2] Nuno Silva[1] Nelma Moreira[1] Simão Melo de Sousa[3*]

[1] Departamento de Ciência de Computadores / LIACC-UP
Universidade do Porto, Portugal
[2] Departamento de Informática / CCTC
Universidade do Minho, Braga, Portugal
[3] Departamento de Informática / LIACC-UP
Universidade da Beira Interior, Portugal

Abstract

This paper presents a brief description of the current work on a tool that analyses temporal behaviour of Ada/RavenSPARK programs. The approach takes as a basis two previous publications that introduce innovative methods in the field of verification of real-time systems. The development of a tool that automatically generates models (timed automata) from Ada/RavenSPARK source code and uses the Uppaal model checker to verify timing properties is discussed.

1 Introduction

New requirements arise from the continuous evolution of computer systems. Processing power alone is not sufficient to satisfy all the industrial requirements. For instance, in the context of critical systems, safety and reliability aspects are fundamental [1]: it is not enough to merely provide the technical means for a set of tasks to be executed; it is also required that the system (as a whole) can correctly execute all of them.

The focus of this paper is precisely the reliability of safety-critical systems [2, 3]. These are usually real-time systems that differ from traditional ones in the sense that they add to traditional reliability requirements the intrinsic need to ensure that tasks are executed within a well-established time scope. For such systems, missing these timing requirements corresponds basically to a system failure. Consequently, an increasing need rises for rigorous analysis in order to enable software designers to create systems that will operate as intended in a real-world environment. Based on this specific requirement the purpose of this paper is to introduce an *automatic* translation tool, able to generate temporal models of programs.

The goal of our tool is to provide an *automated* process for translating relevant control flow information directly from source code, complemented by annotations that further specify the real-time behaviour of programs, into a specification language that can be interpreted by model-checking tools, thus modelling the program's timing behaviour and allowing its timing properties to be verified.

[*]This work is partially supported by the Rescue Project (PTDC/EIA/65862/2006), POSI program and EVOLVE (Projecto N°1621 QREN - SI I&DT), funded by FCT (Fundação para a Ciência e Tecnologia) and QREN (Quadro de Referência Estratégico Nacional).

2 Context

Ada and the Ravenscar profile. The Ada language has been widely used in developing high integrity and real-time applications mainly due to the language's subsets of deterministic constructs designed to ensure full analysability of the code.

The Ravenscar Profile is a subset of the Ada tasking model, restricted to meet the real-time community's requirements for determinism, schedulability analysis and memory-boundness. This profile eliminates nearly all of the Ada tasking constructs that contain implementation dependencies and unspecified behaviour, making it consistent with the use of tools that allow the static verification of program properties. This approach towards a deterministic concurrency model is achieved through the existing definition of pragma `Restrictions` and other configuration pragmas [4][5].

The RavenSPARK subset. Although the Ravenscar profile does not specify any sequential aspects of the language, many verification systems that focus on the sequential part of Ada have been developed. One of the most important is Spark [6], a formally-defined language based in Ada and intended for the development of high integrity software. It consists of a highly restricted, well-defined subset of the Ada language that uses annotated meta information (presented in the form of Ada comments) that describes desired component behaviour and individual runtime requirements. Thus, combining the best aspects of the Ravenscar profile with Spark provides the means to construct highly-reliable concurrent programs. This gives rise to another subset, RavenSPARK [7], which includes another set of restrictions and additional annotations derived from the Spark language, allowing the construction of programs which comply with the Ravenscar profile and retain the analysis facilities of sequential Spark.

The Uppaal tool is a modeling application developed at the universities of **Upp**sala and **Aal**borg, based on networks of timed automata [8]. The tool offers simulation and verification functionality based on the model checking of a subset of TCTL logic [9]. Uppaal is particularly suitable for modelling timed behaviour. Since the model checking engine[10] is independent from the GUI, both visual and textual representations of timed automata can be used for the verification tasks. This is particularly interesting when Uppaal is used in cooperation with other tools. Timing requirements (target properties to be checked) can be specified using the editing facilities of the GUI, or separately in a file.

3 Motivation and Related Work

The verification of the timing properties of critical real-time systems has been the basis of extensive research. Although different approaches exist to this problem, the main common goal seems to be an automated verification process, based on an algorithm capable of analysing the source code of a concurrent real-time program and performing timing properties verification. Next we give a short description of the concepts/references in which we base our approach.

The Deadline primitive. Fidge et al. in [11] introduced a set of real-time programming constructs that enable timing constraints to be directly expressed in a natural and explicit way. The most important of these is the deadline command - a simple statement that expresses upper timing bounds. It accepts an absolute time-valued expression and requires the current time to be no later than this value when the statement is reached. Complemented by the **delay until** statement, available in Ada since its 1995 revision, these constructs enable *all* the timing constraints to be unambiguously expressed. Originally proposed as an additional annotation to the Spark subset of Ada, it has become an important part of a set of annotations that aim at facilitating the verification of timing properties through static analysis of source code.

Temporal Annotations and the Ravenscar profile. Although the main goal of defining the Ravenscar profile was to enable the verification of concurrency aspects of a real-time application, it does not itself directly represent all the primitives necessary to fully specify real-time behavior. In [12] and [13], A. Burns and T.-M. Lin observe that, due to the lack of expressiveness most programming languages possess for expressing timing properties, a collection of annotations can be used to represent and verify temporal requirements. Next we give a short description of the annotations required (note that except for the first and last two, all annotations have a single integer parameter that refers to a time related value):

- **priority** - Indicates the priority of a task or protected object.
- **initialization_deadline** - Indicates the upper bound time value in which task initialization must occur.
- **initialization_latency** - Indicates the lower bound time value in which task initialization must occur.
- **period** - Indicates the period of a periodic task.
- **sporadic** - Indicates the minimum inter-arrival time of a sporadic task.
- **deadline (1)** - Specifies the relative deadline of a task on each invocation/iteration (related to each loop cycle, present only in the task's specification block).
- **deadline (2)** - Specifies the absolute deadline of a task at some defined point of its execution (present only in the task's body).
- **guarantee** - Expresses which sub programs of a protected object will be used by the task.
- **rely** - Expresses which sub-programs of a protected object will be used by other tasks.

The last two annotations express the allowable interference between tasks that share a protected object. They can also express that a task relies on other tasks to *not use* a sub-program (through the **not_op** key), as well as the concept of temporal validity (through the **freshness** key), that indicates the upper bound time value of a protected object's sub-program periodic calls. Except for the deadline annotation, which can also appear in the body, all proposed annotations can only be used in the specification part of an Ada/RavenSPARK program. The delay statement does not need an annotation since it is directly represented by the "delay until" statement in Ada/RavenSPARK programs [13].

With the introduction of the RavenSPARK subset [7] into the Spark language, a new space for reserved words and annotations is created and taken into account by Spark tools. Through the use of the reserved word *declare*, these annotations can be provided as a complement to Spark annotations to specify the real-time behavior of an application.

ASIS and the Avatox application. The Ada Semantic Interface Specification (ASIS) is an interface between an Ada environment as defined by ISO/IEC 8652 (the Ada Reference Manual) and any tool requiring information from this environment. An Ada environment includes valuable semantic and syntactic information [14]. ASIS is an open and published callable interface which gives CASE tool [15] and application developers access to this information. Through queries to the Ada compilation unit, the ASIS interface provides means for retrieving all of its semantic and syntactic content in the form of an abstract syntax tree, that contains abstractions for all Ada components in an useful hierarchical structure.

Avatox [16] is an application that traverses one or more Ada compilation units and outputs the ASIS representation of the unit(s) as a XML document. The resulting XML document(s) can then have an XSL stylesheet [17] applied in turn. Given that the Avatox XML representation of the source code comprehensively represents the content and layout of the source code, many methods for extracting and processing this information become available.

Automatic Verification of Real-Time Systems. Joel Carvalho [18] in his master thesis, introduced a new tool to verify HTL programs. As it is known, the HTL programming language is used to coordinate and specify temporal behaviour between programs written in different languages and does not have built in primitives to develop real-time programs. In this work the idea is to extend the verifying tool-chain offered by the HTL suite. This tool-chain is based on an automated translation tool called HTL2XTA also developed during the master thesis work, and that performs an automatic translation from HTL source code to Uppaal automata.

The HTL tool-chain performs static analysis of programs during the compilation stage. To complement this, Joel Carvalho's approach allows the temporal analysis of the behaviour of tasks. The Uppaal suite enables model analysis, execution simulation (through state transitions) and property verification to be performed by an integrated model checker (verifyta). To understand how this translator and tool-chain works the reading of the introduction paper and the master thesis document [19] are recommended, since all the algorithm's details are explained there.

This approach has provided a useful inspiration for the development of our tool. Although the expressiveness of Ada can by no means be compared to HTL, which is a much more restricted and focused language, extensive study of the translation algorithm proposed by the HTL2XTA tool chain has provided some hints of how the translation of Ada programs can be achieved.

4 The Translation Process

Description and Goals. Bearing in mind the methodologies described in [11],[12] and [18], which are the main influences in this work, we introduce a tool able to automatically:

a) extract Uppaal models directly from Ada code;

b) infer timing properties to be checked with the Uppaal model checker.

In fact, all investigation previously done shows that this translation is not so simple. Since the Ada language provides a large set of instructions, the first approach was to restrict the tool only to programs that are compliant with the Ravenscar profile. Moreover, another conclusion immediately drawn is the importance of the annotations proposed in the previous works, since they are relevant to the success of the tool. Nevertheless, other helpful annotations may be introduced once shown that they are determinant in the translation process.

The translation method proposed is accomplished in three phases. The first step, with the help of the Avatox tool, is to generate one or more XML files containing the abstract syntax tree (AST) of a program's source code (.adb and .ads). This choice is justified by the existence of several methods and tool facilities available to process XML files. In the next step, an algorithm to generate control flow graphs (CFG) [20][21] from the existing XML files is developed. Finally, based on the control flow graph's information and on all annotations present in the source code (also extracted from the XML files), the goal is to analyse and transform the generated CFG into an Uppaal model that contains the program's structure as well as its temporal requirements.

In order to enhance the funcionalities of this tool, an algorithm that automatically infers timing properties based on timed automata theory is provided. These properties are all related with some of the program's specifications and typically specify the system's temporal requirements, such as the timing bounds imposed on some instructions or blocks of instructions, like delays and deadlines.

Algorithm Details

In this section we give some details from the current state of the tool, focusing on the details of CFG generation process.

As previously described, Avatox is based on the ASIS platform, so the XML tags present on generated files are also based on the type set used by ASIS. Hereupon, the XML files can be defined as a set of several terms organized in a tree structure representing the program's syntax.

Consequently, the algorithm is basically a traversal over the tree structure and a switch case for processing each relevant tag (ASIS Element) found. Relevant tags are those that correspond to simple transitions i.e. an instruction followed by another, an *if statement* or a *loop statement*, as described below.

Simple instructions. This is the simplest case found on a CFG. When a simple instruction is followed by another one, the correspondent CFG is simply a node linked to another (Fig. 1.).

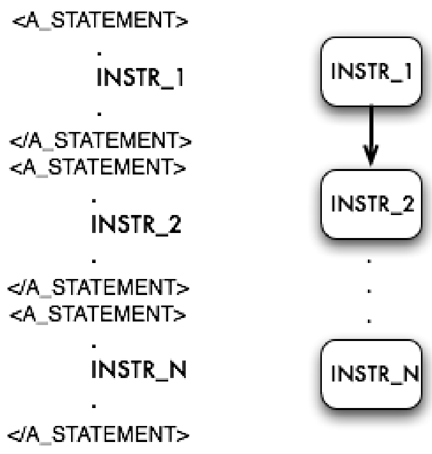


Figure 1: Translation of a simple instruction followed by another.

If statement. In terms of control flow, an *if statement* specifies the possibility of program's execution taking one of two paths, depending on the evaluation of one condition. The resulting CFG must specify this option. Then, when a XML tag with an if statement is found, links to two new nodes are added to CFG, one for each possible path. This scenario is shown in Figure 2.

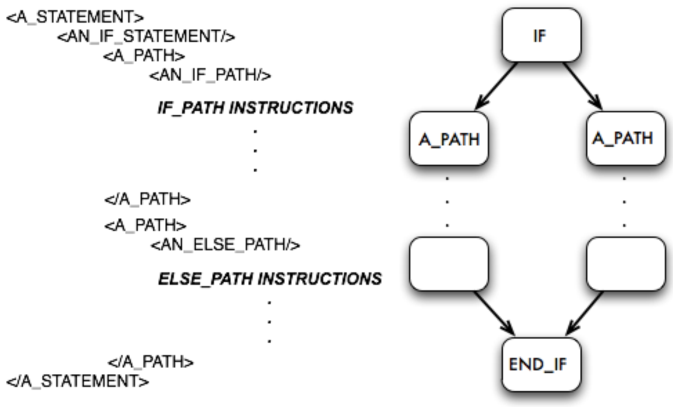


Figure 2: Translation of if statement

Loop Statement. There are two kinds of loops, with or without a condition. Loops without a condition are typically infinite loops, and in Ada/Ravenscar are always present at the end of a task body statement (due to the profile's restrictions [4]). For this kind of loop we just need to add a link between the last loop instruction and the first one. On the other hand, in addition to the link previously described, conditioned loops have a link between the node that contains the loop's condition evaluation and the node after the loop's body end. This way the case where the initial evaluation fails is covered. Figure 3 shows the first scenario, i.e. infinite loops.

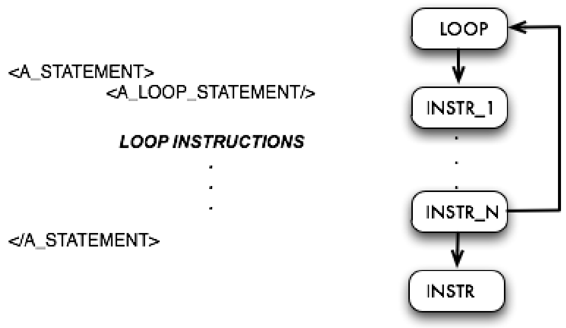


Figure 3: Translation of loop statement

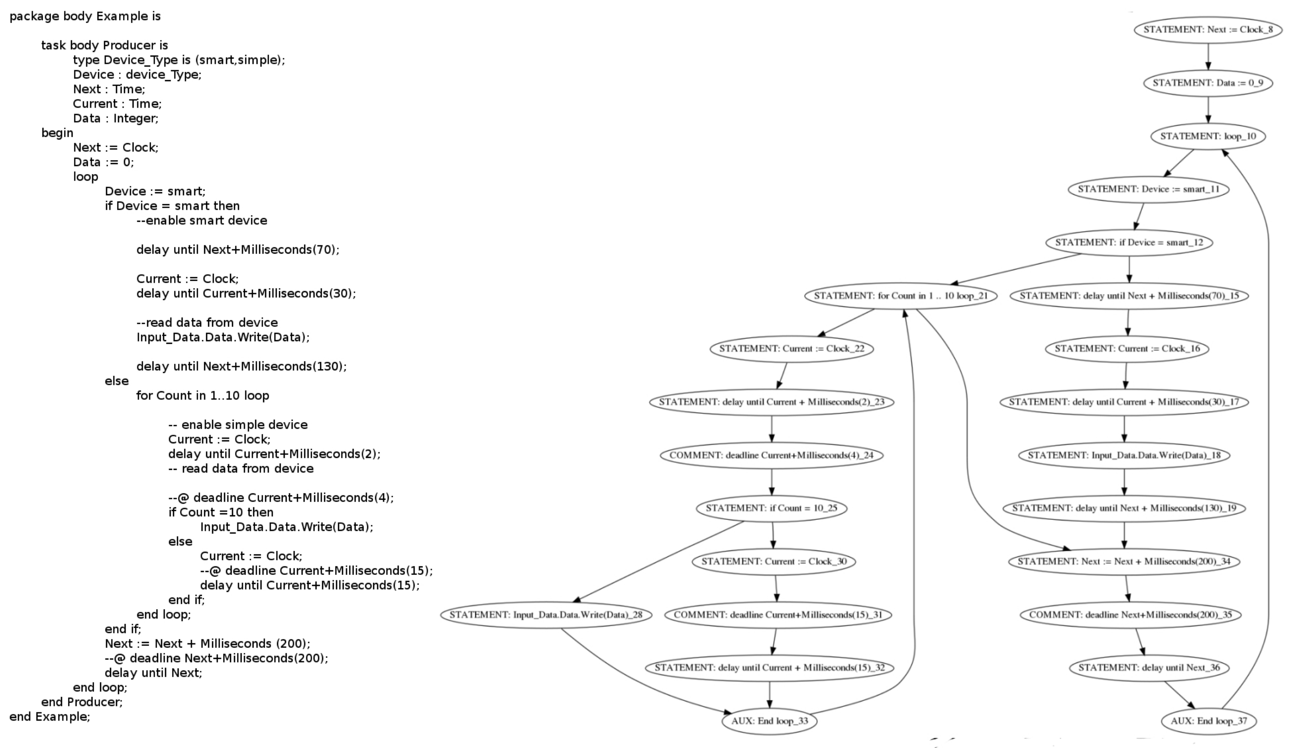
The current version of the translator (developed in Java) automatically generates a program's CFG. There are still difficulties in translating some ASIS elements, but at this point they are not that relevant since all the main control flow information is contained in the generated CFG.

The next example shows the translation process from the Ada code to the correspondent Uppaal model focusing the intermediate step of control flow graph generation. The XML file generated by Avatox is omitted in the following set of figures since its verbosity does not enable the extraction of a fragment capable of illustrating the XML code representative of an Ada program.

Experimental Validation

Program description. The following Ada program was directly inspired by the test case used in [12] and [13], and serves to illustrate the current status of the tool's performance. It intends to simulate a system that reads data from one of two possible sensors, a simple sensor and a smart sensor, producing and writing new data in a protected object. The difference between the two sensors lies on the reliability of the values read: one provides a reliable value for each reading while the other requires the sensor to be read ten times in order to get one single reliable value. The system also requires that timing bounds be specified for some situations. For instance, it is specified that the protected object's data must be updated with a new value every 200 milliseconds and that the smart sensor should provide a reading 30 milliseconds after being enabled. Due to this example's purpose, we will just highlight the fact that the protected object's data must be updated every 200 milliseconds or earlier.

As seen in Figure 4, the control flow information of the algorithm is partially described. In order for the final result to be presented, some instructions (nodes) that belong to the generated CFG have been suppressed due to space reasons.



```
package body Example is

    task body Producer is
        type Device_Type is (smart,simple);
        Device : device_Type;
        Next : Time;
        Current : Time;
        Data : Integer;
    begin
        Next := Clock;
        Data := 0;
        loop
            Device := smart;
            if Device = smart then
                --enable smart device

                delay until Next+Milliseconds(70);

                Current := Clock;
                delay until Current+Milliseconds(30);

                --read data from device
                Input_Data.Data.Write(Data);

                delay until Next+Milliseconds(130);
            else
                for Count in 1..10 loop

                    -- enable simple device
                    Current := Clock;
                    delay until Current+Milliseconds(2);
                    -- read data from device

                    --@ deadline Current+Milliseconds(4);
                    if Count =10 then
                        Input_Data.Data.Write(Data);
                    else
                        Current := Clock;
                        --@ deadline Current+Milliseconds(15);
                        delay until Current+Milliseconds(15);
                    end if;
                end loop;
            end if;
            Next := Next + Milliseconds (200);
            --@ deadline Next+Milliseconds(200);
            delay until Next;
        end loop;
    end Producer;
end Example;
```

STATEMENT: Next := Clock_8
STATEMENT: Data := 0_9
STATEMENT: loop_10
STATEMENT: Device := smart_11
STATEMENT: if Device = smart_12
STATEMENT: for Count in 1 .. 10 loop_21
STATEMENT: delay until Next + Milliseconds(70)_15
STATEMENT: Current := Clock_22
STATEMENT: Current := Clock_16
STATEMENT: delay until Current + Milliseconds(2)_23
STATEMENT: delay until Current + Milliseconds(30)_17
COMMENT: deadline Current+Milliseconds(4)_24
STATEMENT: Input_Data.Data.Write(Data)_18
STATEMENT: if Count = 10_25
STATEMENT: delay until Next + Milliseconds(130)_19
STATEMENT: Current := Clock_30
STATEMENT: Next := Next + Milliseconds(200)_34
STATEMENT: Input_Data.Data.Write(Data)_28
COMMENT: deadline Current+Milliseconds(15)_31
COMMENT: deadline Next+Milliseconds(200)_35
STATEMENT: delay until Current + Milliseconds(15)_32
STATEMENT: delay until Next_36
AUX: End loop_33
AUX: End loop_37

Figure 4: CFG generated by the current version of tool

As previously described the final step of the tool's translation algorithm is to analyse and transform the generated CFG in order to reach an Uppaal model. In detail, it is possible to state that almost all of the CFG's nodes are present in this model, and these are all the nodes that directly influence the timing behaviour of the program. Moreover, this model enables us to observe the full temporal behaviour specified in the system's requirements. Next we give the Uppaal model of the Ada program above described.

Comparing the model obtained with the one presented in Figure 6[1] it is possible to observe obvious similarities and some differences. The differences ride from the fact that this tool provides an automated process of translating Ada source code that follows closely the execution path of the program. Ada code instructions are processed with only the necessary context information, resulting on a non-optimized timed automata that behaves equivalently to an optimized one (with the drawback of additional time needed for property verification). Verification of some of the program's main timing properties[13] is successfully achieved:

```
A[] not deadlock

A[] Producer.Next <= 200
```

Similar tests have been undertaken, using the same approach on different programs with successful results. However, the complexity of the translation process tends to increase with the size of the program. Real-time systems are often composed by many modules, and when processing communication aspects in large programs, several difficulties arise. In order to

[1]Figure imported from [13], page 120, Figure 8.

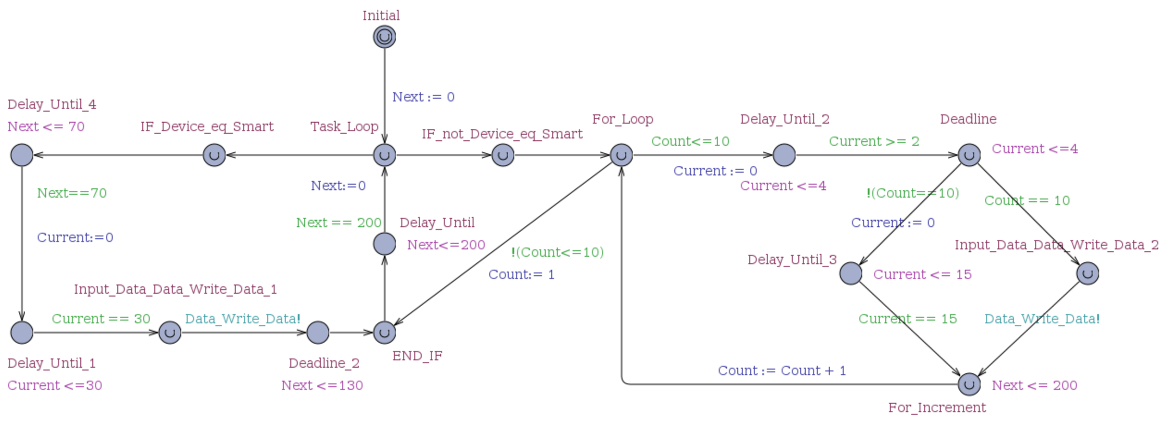


Figure 5: Uppaal model generated by the tool

accurately simulate the program's behaviour the concurrency model has to be well taken into account. For instance, when modelling Ravenscar compliant programs, task priorities and scheduling policies must have a precise equivalence in the generated model. Due to Uppaal's limited input language, this and other similar problems present difficulties which will be addressed in the near future.

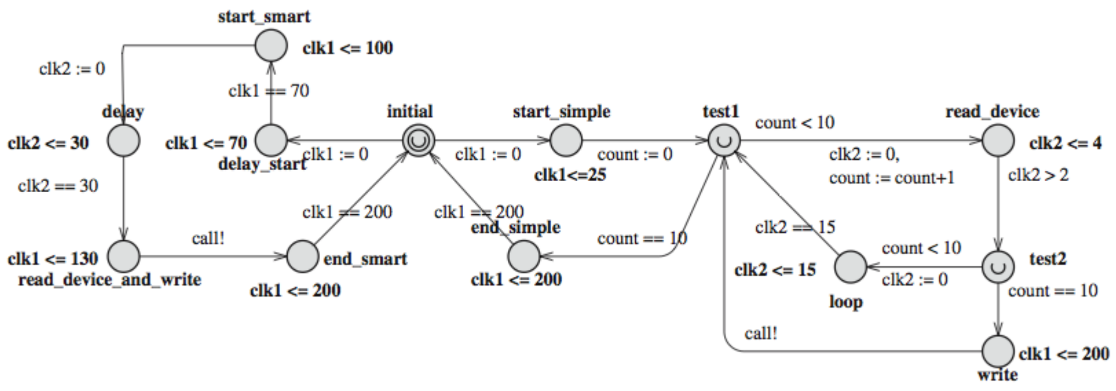


Figure 6: Uppaal model developed by Alan Burns presented in [13]

5 Conclusion and Future Work

This paper has described an automatic translation method to generate temporal models of Ada real-time programs.

The model's construction is based on two essentials strategies: the processing of the program's control flow information and the use of annotations to further specify the program's temporal behaviour.

Currently, two approaches are being explored with the common goal of improving the tool's performance. One consists on refining the process of gathering the control flow information of a program, which will result in a more accurate and well-specified model. The other is based on the process of code reconstruction and translation into a specification language capable of being interpreted by model-checking tools. The idea is to compare the results obtained by both methods, verifying what kind of properties each one will be able to check. We intend to combine the best of both methods in order to improve the tool's results.

Presently the main concern has been the performance of different experimental tests. These tests are similar to the one described in this paper and the experimental validation undertaken also consists in observing and comparing the results obtained with those of other publications [12] [13]. We think that these test cases are a good way to prove the tool's reliability. A formal verification of the tool is also one of the project goals once its performance stabilizes with acceptable results.

Another issue to be addressed in a near future is the completion of the algorithm that automatically infers properties. We believe this step is very important since the generation and verification of properties that cover most of the program's timing requirements will improve the system's reliability.

Summing up, this tool may improve the current set of alternatives to analyse real-time systems. Moreover, we hope this work represents an open door to further publications on this topic. Indeed, the main goal of this project is to offer the industry an useful and reliable tool capable of improving the quality and security in the software development area.

References

[1] J. Rushby, "Formal methods and their role in the certification of critical systems," tech. rep., Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop, 1995.

[2] S.-T. Levi and A. K. Agrawala, *Real-time system design.* New York, NY, USA: McGraw-Hill, Inc., 1990.

[3] P. J. Stankovic and A. Stankovic, "Real-time computing," 1992.

[4] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the ada ravenscar profile in high integrity systems," *Ada Lett.*, vol. XXIV, no. 2, pp. 1–74, 2004.

[5] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, and P. Leroy, *Ada 2005 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1 (Lecture Notes in Computer Science).* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[6] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.

[7] S. Team, "Spark examiner - the spark ravenscar profile," pp. 1–73, 2008.

[8] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," 2004.

[9] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004* (M. Bernardo and F. Corradini, eds.), no. 3185 in LNCS, pp. 200–236, Springer–Verlag, September 2004.

[10] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model checking.* Cambridge, MA, USA: MIT Press, 1999.

[11] C. Fidge, I. Hayes, and G. Watson, "The deadline command," 1998.

[12] A. Burns and T.-M. Lin, "Adding temporal annotations and associated verification to ravenscar profile," in *Reliable Software Technologies—Ada-Europe 2003* (J.-P. Rosen and A. Strohmeier, eds.), vol. 2655 of *Lecture Notes in Computer Science*, pp. 80–91, Springer-Verlag, 2003. Ravenscar Profile, Model Checking, UPAAL, SPARK.

[13] A. Burns and T. M. Lin, "An engineering process for the verification of real-time systems," *Form. Asp. Comput.*, vol. 19, no. 1, pp. 111–136, 2007.

[14] A. W. Group, "An interface to the ada95 compilation environment," June 2010.

[15] S. Jarzabek and R. Huang, "The case for user-centered case tools," *Commun. ACM*, vol. 41, no. 8, pp. 93–99, 1998.

[16] M. A. Criley, "Avatox (ada, via asis, to xml)," Aug. 2007.

[17] W. W. W. Consortium, "Xsl transformations (xslt) - version 2.0," tech. rep., 2007.

[18] J. Carvalho and S. Melo, "Verificação de modelos de programas htl," 2009.

[19] J. Carvalho, "Verificação automatizada de sistemas de tempo-real críticos," Master's thesis, Universidade da Beira Interior, 2009.

[20] R. Fechete, G. Kienesberger, and J. Blieberger, "A framework for cfg-based static program analysis of ada programs," in *Ada-Europe '08: Proceedings of the 13th Ada-Europe international conference on Reliable Software Technologies*, (Berlin, Heidelberg), pp. 130–143, Springer-Verlag, 2008.

[21] E. Moretti, G. Chanteperdrix, and A. Osorio, "New algorithms for control-flow graph structuring," in *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, (Washington, DC, USA), p. 184, IEEE Computer Society, 2001.