A Tool for Automatic Model Extraction of Ada/SPARK Programs

Nuno Silva   Nelma Moreira   Simão Melo de Sousa   Sabine Broda

Technical Report Series: DCC-2011-2
Version 1.0 October 2011

U. PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

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

Nuno Silva[1] Nelma Moreira[1] Simão Melo de Sousa[2] Sabine Broda[1]

[1] Departamento de Ciência de Computadores / LIACC-UP
Universidade do Porto, Portugal
[2] 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 behavior of Ada/RavenSPARK programs and, with the use of a translation algorithm, outputs an Uppaal system that simulates the program's control flow.

The focus will be the translation algorithm, it's implementation and syntactic scope, along with results and difficulties encountered during the development process.

1 Introduction

The use of model-checking [1] as a means to perform software verification appears to us currently as one of the most prominent methods to achieve formal verification of software correction. Although an ambitious and difficult task, the correct syntactic translation and modeling of the source code would come as a very important aid in achieving this goal.

This report describes the recent developments in a tool that aims at translating and verifying temporal behavior of real-time applications. The starting point is the Ada/Ravenscar profile [2], a subset of the Ada tasking model restricted to meet the real time community's requirements for determinism, schedulability analysis and memory-boundness. The existence of the Ada Semantic Interface Specification (ASIS) [3], an interface for retrieving syntactic and semantic information from the Ada environment, and of a number of XML-based tools for processing this information make it an ideal candidate for this translation process. On the other end, the UPPAAL application [4], based on networks of timed automata, offers the simulation and functionality needed for modeling an application's temporal behavior.

We'll focus mainly on the second stage of development of this tool, which has been the development and implementation of a process to achieve a solid syntactic translation of a program's source code into a temporal system that correctly models the program's control flow.

2 The deadline annotation

Modeling the temporal behavior of a program requires the programming language to have a certain expressiveness in order for the timing bounds to be set in the source code. Although

Ada has a very complete set of concurrency constructs, the use of additional annotations can further increase the precision of these timing bounds.

Fidge et al. in [5] 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, which expresses a lower timing bound, 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 [6] 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.

Due to this fact, the deadline has become the only annotation required for modeling a program's control and time flow. A further set of annotations is being explore with the goal of property verification, which will be used after the translation process as a means to express and verify the program's timing restraints.

3 Tool work flow description

We can divide the tool's work flow in three stages, each with it's particular scope. In this section we aim at providing the reader with a brief explanation of the tool's implementation by describing each stage. The following diagram synthesizes the tool's work flow, giving a broad perspective of the translation process.

Stage 1: Preprocessing input files. The first stage relies heavily on the Ada Semantic Interface Specification (ASIS) as a means to transform the Ada source code into a format suitable for translation. To achieve this transformation two applications are used:

- Avatox [7], an application that traverses one or more Ada compilation units and outputs the ASIS representation of the unit(s) as a XML document. 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.
- XALAN [8], a XSLT [9] processor for transforming XML documents. It is used to further optimize the ASIS hierarchical representation of the Ada source code by deleting a number of tree levels and moving import attribute information.

After these two preprocessing steps the Ada source code has been converted into a suitable format for translation. Our Java application RAST (RAvenScar Translator) will then use Java API's to manage the information contained in the XML input files.

Stage 2: Preprocessing source code information. In this stage we collect, organize and store all the information necessary to perform the translation and modeling of the Ada/Ravenscar program. We start by organizing the input files in groups that correspond to Ada packages. Each of these groups will then be transformed into a data structure we call an Ada module. This will be done in two steps. First, the specification parts of each package will be processed: a loop will iterate each ASIS element present in the specification file of

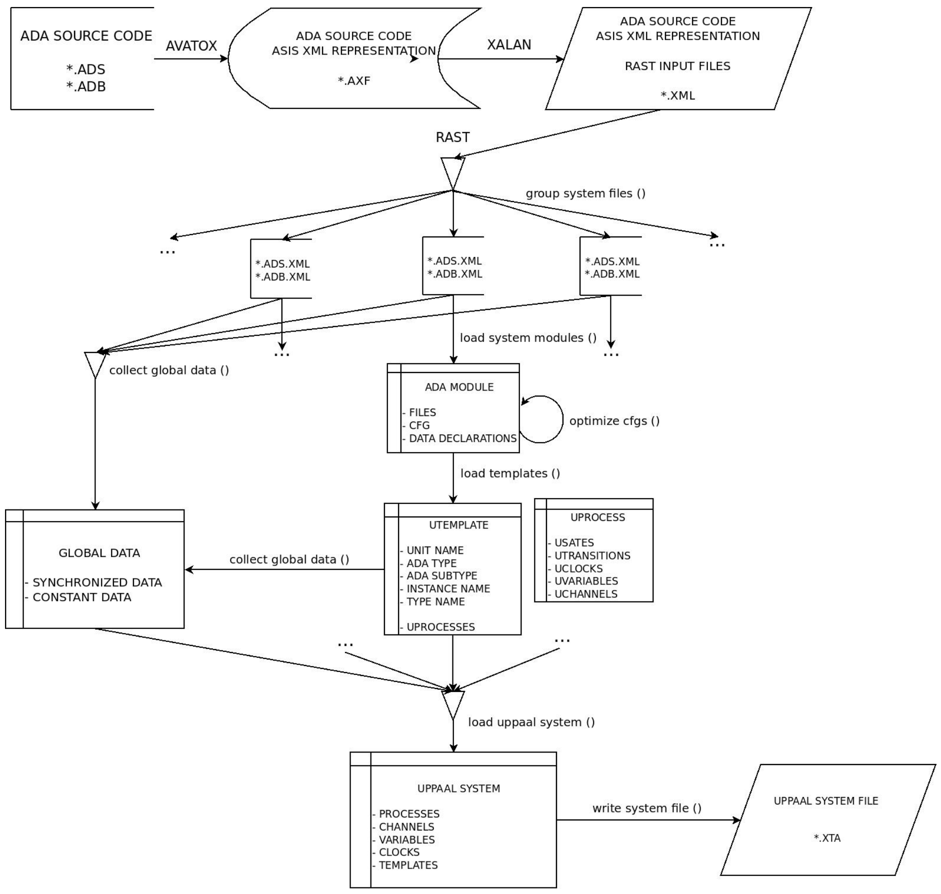


Figure 1: Tool workflow diagram.

the package and a switch-case will filter and store relevant package, variable and function information. Next, using the information collected, we process the package's body file: in a similar way, we loop through the ASIS elements and use a switch-case to filter additional declaration data and to build the control flow graphs that correspond to the package's body work flow. Here, the ASIS elements are processed in several levels: the first level focuses on the main Ada constructs in the Ada package (tasks, protected objects, procedures and functions), which will probably result in Uppaal processes after translation. For each of these elements, a new traversal is created which generates the CFG (control flow graph) [10] of the construct. Of course, some of the body's statements, such as loops and conditional expressions, require a recursive treatment, giving rise to new traversals that are automatically coupled with the CFG to be generated.

After these steps, performed for each Ada package, we'll have a set of Ada modules that will contain all the information needed for translation. To end this preprocessing stage we run a series of routines that optimize the CFG's in order for them to contain only the relevant modeling information and perform a final sweep of the input data to gather global system data for the purpose of system synchronization and package dependency inspection.

Stage 3: Translation and output. The final step consists in the actual translation. Each of the CFG's contained in each of the Ada modules is translated to what we call a Uppaal template (which has a different meaning of the template type used in the uppaal input language), that contains the timed automata that represents the construct's control flow plus all the declaration data associated to it, representing closely an uppaal process.

This is accomplished by traversing each CFG and converting each node to an uppaal state and each edge to an uppaal transition. Of course, in some cases, translation cannot be performed locally, that is, we need to inspect more than one node and variable information in order to translate a single transition/state. However, an effort has been made to make each translation as local as possible. Uppaal clocks and other variables are translated by inspecting the module's declaration data and uppaal channels by searching the synchronized nodes of the CFG. After the translation, each template will contain an uppaal process plus the necessary information needed for it to be instantiated in an uppaal system.

Finally, after we run a final sweep of all templates to gather all necessary global data and the uppaal constructs are assembled in a single uppaal input file that contains all the uppaal system data.

4 Algorithm Details

In this section we'll describe the current state of the tool, focusing on the details concerning how the most relevant Ada constructs are translated to the uppaal input language. Since the main focus of this tool is to model the temporal behavior of concurrent Ada/Ravenscar applications [11], the translation algorithm focuses it's attention on the specific Ada constructs [12] that relate directly to these characteristics. To further illustrate how the translation is done we will show the input and output results taken from two Ada/Ravenscar programs used to test the tool.

The first has been inspired by the test case used in [13] and [14], and serves to illustrate how ordinary sequential commands contained in an Ada task body are translated. 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.

The second has been taken from [15] and consists on a simple stopwatch program. In this section we'll use parts of this application to exemplify specific translation methods while in the next section we'll describe this system's functionality in full to experimentally validate the tool's translation algorithm. Next we give an overview of the translation process for the most relevant ADA constructs present in typical Ada/Ravenscar based applications.

Regular sequential flow. Modeling simple sequential instructions that do not relate to the program's concurrent and timing behavior, such as single instructions, loops or conditional expressions, is done in a very straightforward way through the use of a branch generating algorithm normally used in CFG creation. However it's important to notice that this must be done so that the model's behavior is the same as the program's, otherwise we would get an inaccurate simulation of the program's behavior.

The following example, part of the sensor system referred above, shows exactly this; we have a for loop that increments the variable count 10 times. The count variable is evaluated in an if expression and while it takes values from 1 to 9 the program stalls for 15 milliseconds and the count variable is incremented. When the variable takes the value 10 a protected procedure is called and the loop exits. Not modeling correctly loop variables and almost any other conditional variable represents a flaw in the model and invalidates any temporal analysis. This represents a major difficulty in accomplishing such a translation due to the extension of ADA's data types and syntax variations.

Thus far the main part of ADA's syntax is encompassed by the tool for translation, such as if expressions, for/while loops that evaluate an integer variable, among others, which is enough to perform a complete translation of simple programs.

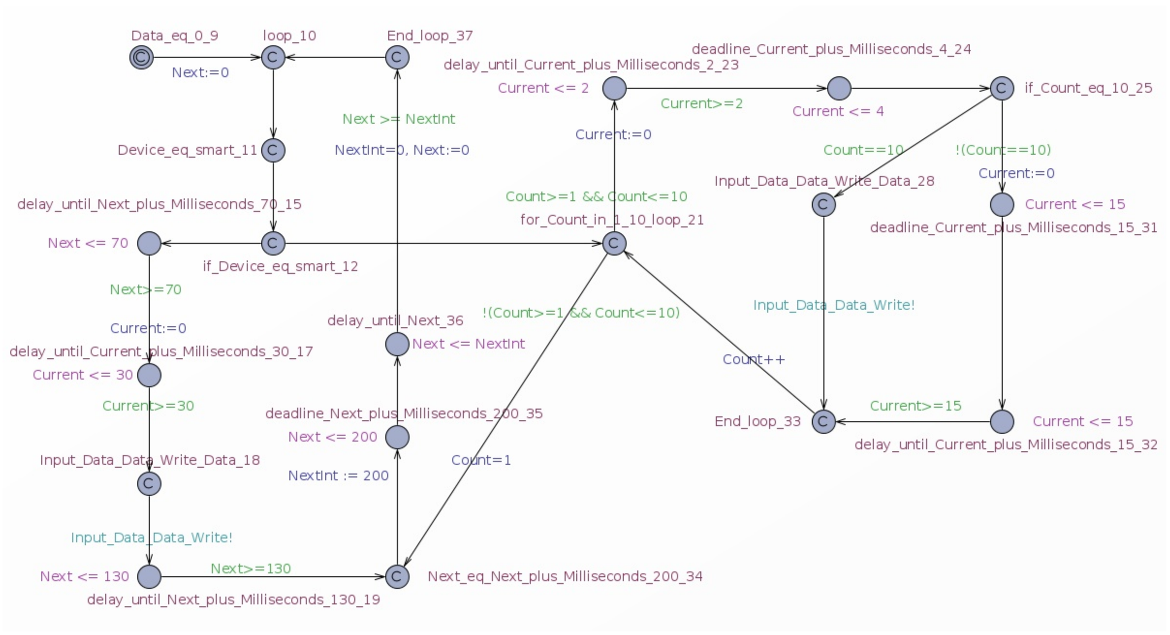
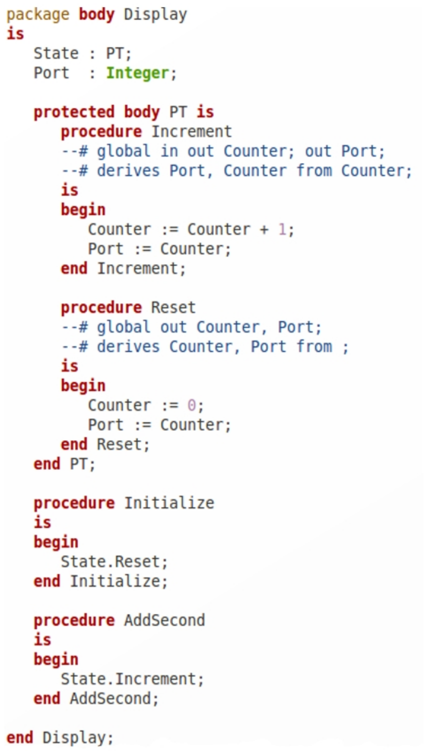


Figure 2: Ada task containing common sequential instructions.

Tasks. Ada tasks are in charge of most of the activity that is performed by an Ada/Ravenscar application. As happens with protected objects, Ada tasks may be declared as types or single tasks, whose type is referred to as anonymous. When declared as a type, a task can be instantiated any number of times with different arguments. Modeling a task type requires that a template be created for the task. This template is translated to an Uppaal process and is instantiated accordingly with the number of calls and corresponding arguments present in the source code. A task's body is translated in a straightforward way and, as is a requirement of the ravenscar profile, cannot terminate, so we always find at the end of a task's body an infinite loop that ensures this requirement.

Protected Objects. The protected object's translation method is the only Ada construct which requires a different treatment. A protected object is translated into an automata that models all the protected entries, procedures and functions in the protected object's body. Through Uppaal's synchronization primitives [16], a channel listens at the start of each procedure/function/entry and when a signal arrives it's body is executed. Since all of the states are committed, the execution of the body happens instantaneously, which simulates the atomicity inherent to a protected object's methods.

Procedures and Functions. For a program's behavior to be correctly modeled an Uppaal process must be created for every procedure or function. This happens because any procedure or function may have, in it's body, temporal primitives or calls to other functions that condition the program's behavior. To describe this situation we'll use part of the stopwatch



```
package body Display
is
   State : PT;
   Port  : Integer;

   protected body PT is
      procedure Increment
      --# global in out Counter; out Port;
      --# derives Port, Counter from Counter;
      is
      begin
         Counter := Counter + 1;
         Port := Counter;
      end Increment;

      procedure Reset
      --# global out Counter, Port;
      --# derives Counter, Port from ;
      is
      begin
         Counter := 0;
         Port := Counter;
      end Reset;
   end PT;

   procedure Initialize
   is
   begin
      State.Reset;
   end Initialize;

   procedure AddSecond
   is
   begin
      State.Increment;
   end AddSecond;

end Display;
```

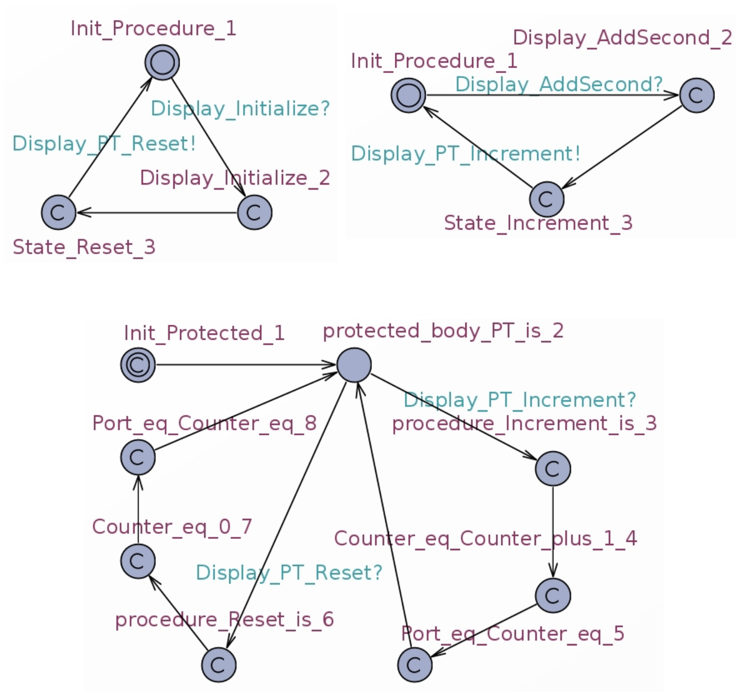
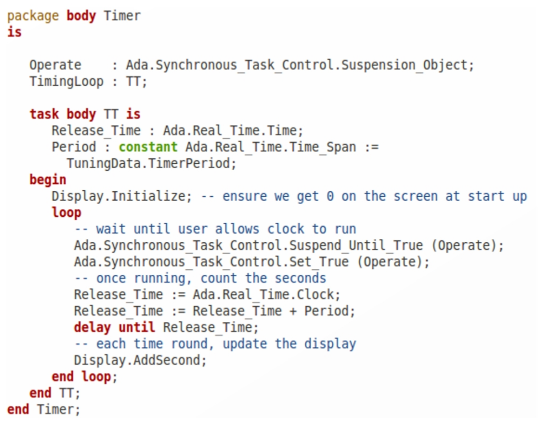


Figure 3: Stopwatch Display package translation.

system taken from a RavenSpark application [15]. The first two automatas correspond to the procedures declared in the package. One initializes the display associated with the stopwatch and the other increments the clock's counter. Within the body of the first procedure a protected call is made to procedure Reset which will cause the counter to be set to zero. This is a common case in Ada/Ravenscar applications referred to as protected refinement, where protected calls are encapsulated in procedure calls.

We can easily observe that if this procedure was not modeled in such a way the protected object's procedures would never be executed and, although in this specific case it does not happen, important control flow information could be missed thus compromising the system's correct simulation of the original application.

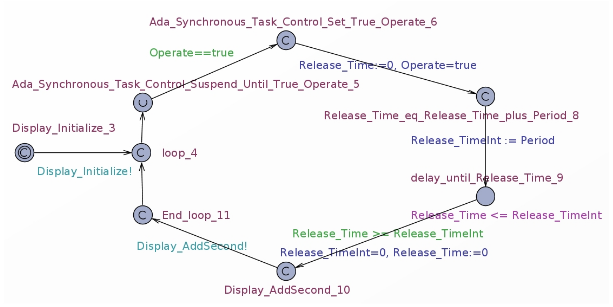
Suspension Objects. Ada's suspension objects [12] have been so far the only Ada concurrency construct to be processed by the tool. This construct behaves much like a binary semaphore, with the normal blocking and querying operations associated with it. Although it is not the only one that needs such a treatment, it's translation is vital for the control flow of the model to resemble that of an Ada program. This translation however is quite simple: to represent the internal state of the suspension object a boolean variable is created and initialized with a false value. The atomic operations that change the suspension object's state are modeled with simple attribution's to this boolean variable. The only blocking operation this construct provides, the *Ada.Synchronous_Task_Control.Suspend_Until_True* procedure, is modeled by simply placing a guard on the ongoing transition in the model that tests if the variable's value is true, in which case it proceeds normally . If not, the model's flow will stall in the current state until the variable's value is set to true.



```
package body Timer
is

   Operate    : Ada.Synchronous_Task_Control.Suspension_Object;
   TimingLoop : TT;

   task body TT is
      Release_Time : Ada.Real_Time.Time;
      Period : constant Ada.Real_Time.Time_Span :=
        TuningData.TimerPeriod;
   begin
      Display.Initialize; -- ensure we get 0 on the screen at start up
      loop
         -- wait until user allows clock to run
         Ada.Synchronous_Task_Control.Suspend_Until_True (Operate);
         Ada.Synchronous_Task_Control.Set_True (Operate);
         -- once running, count the seconds
         Release_Time := Ada.Real_Time.Clock;
         Release_Time := Release_Time + Period;
         delay until Release_Time;
         -- each time round, update the display
         Display.AddSecond;
      end loop;
   end TT;
end Timer;
```



Ada_Synchronous_Task_Control_Set_True_Operate_6
Operate==true
Release_Time:=0, Operate=true
Ada_Synchronous_Task_Control_Suspend_Until_True_Operate_5
Release_Time_eq_Release_Time_plus_Period_8
Display_Initialize_3
Release_TimeInt := Period
loop_4
Display_Initialize!
delay_until_Release_Time_9
End_loop_11
Release_Time <= Release_TimeInt
Release_Time >= Release_TimeInt
Display_AddSecond!
Release_TimeInt=0, Release_Time:=0
Display_AddSecond_10

Figure 4: Task containing a suspension object.

Variables: Clocks and Time Spans. Given the tool's purpose we identify two variable types as the most important for the translation process: clocks and time spans. Their translation is direct from Ada types *Ada.RealTime.Time* and *Ada.RealTime.TimeSpan* to Uppaal types *clock* and *const int* respectively. Ada's time spans usually contain values used to set task periods: their argument is directly translated to an uppaal *const int* variable.

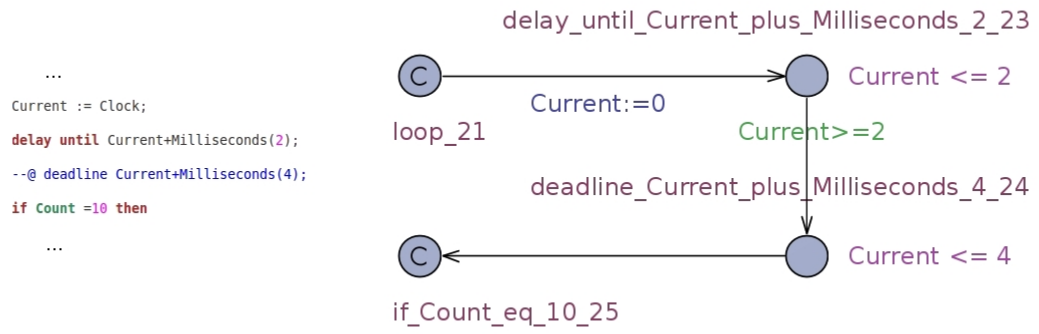
However, clock behavior is different in each language: in Ada these variables are used to store time values through a group of operations such as *Ada.RealTime.clock()*, which retrieves the current internal time value, arithmetic operations, among others. Bounding instructions such as the delay until command will compare Ada's internal real time clock value to values stored in the clocks to control the program's time flow. In Uppaal, clock variables have a different behavior: the time value they hold is incremented automatically by uppaal's model-checking engine and they are questioned by uppaal's invariant and guards, which control the time flow of the automata. Given this fact, clock translation requires a special treatment: regarding clock attributions, we opted to create auxiliary variables in each uppaal model to contain the values that are updated in the Ada source code. When clocks are questioned, for example with a delay until command, the value of the uppaal clock is compared to that of the auxiliary variable. This method allows us to model the time flow in an uppaal automata in a very similar to that of an Ada program.
The main operations regarding these variables are:

- **Attributions**: Attributions to clock variables are translated to uppaal by creating an auxiliary integer variable that will contain the final value of the attribution made in the Ada source code. These auxiliary variables will resemble closely the behavior of Ada's time variables providing bounds for the clock variables to be tested against. The main difference resides on the task's infinite loops: here, instead of letting the time values increase infinitely, as happens in an Ada program, in uppaal we choose to reset the clocks in every loop iteration, keeping the clock time values bounded by the period of the loop. This facilitates translation and eases the burden of testing extremely large time values to the model-checking engine.

- **Reset**: The Ada instruction *<Clock_varname>:= Ada.RealTime.Clock()*, which sets the variable's value with the current time value of Ada's internal clock is translated as simple reset of the uppaal clock variable (*clock := 0*).

- **Delay Until**: In Ada, the delay until semantics provide a lower bound for the program's time flow to proceed. However this presents an incompatibility with Uppaal's time flow constraints: allowing an automata state to have just a lower time bound would lead the model-checking engine to test extremely large time values for that state which will inevitably lead to an erroneous system state. Therefore, in order to model correctly a program we chose to compromise the delay until's semantics and ensure that this lower bound will in fact be an exact value for the time flow to proceed. A translation example will be provided in the following figure.

Annotations: Deadline. Thus far the tool makes use only of annotations present in an Ada package's body. As proposed in [5, 13], the deadline annotation will allow the programmer to fully expressed all the timing bounds that he wishes the program to have. Thus we employ the use of the deadline annotation to express an upper bound on the program's time flow as presented below: the translation to uppaal is made by creating a state in which the invariant expresses the bound that appears as the deadline's argument.



```
    ...
Current := Clock;
delay until Current+Milliseconds(2);
--@ deadline Current+Milliseconds(4);
if Count =10 then
    ...
```

delay_until_Current_plus_Milliseconds_2_23
C
Current <= 2
Current:=0
loop_21
Current>=2
deadline_Current_plus_Milliseconds_4_24
C
Current <= 4
if_Count_eq_10_25

Figure 5: Delay until and Deadline translation.

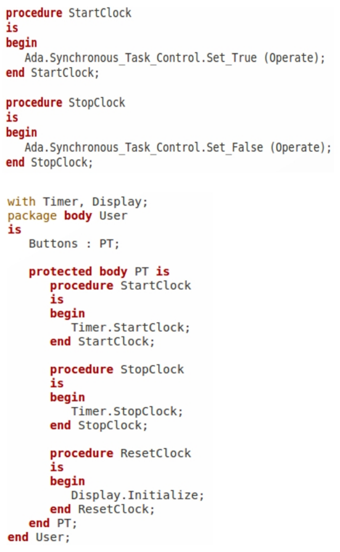
System Synchronization : There are two types of synchronization applied during the translation process. To model protected objects correctly, as has already been stated above, a channel must be created for every protected procedure, function and entry so that, upon invocation, that method will be executed. In a similar way, for all procedures and functions present in Ada packages a new uppaal process is created as well as an uppaal channel that will listen at the start of the procedure/function for it's invocation. For large systems this approach will result in the creation of a reasonable amount of uppaal channels, but for reasons stated above, this must be done so that the program can be correctly translated into an uppaal model.

4.1 Experimental Validation

In this section we'll observe an Ada program chosen from our test cases and describe it's full translation to exemplify the tool's ability to translate and model an operating Ada program. Parts of this program have already been shown in the former section and will be referred to as needed. The tool's utility will then be described by an auxiliary model that will simulate a user's interaction with the program and with examples of timing properties that can aid in proving that the program's timing constraints are being met.

Program description. The following Ada program was taken from the SPARK Ravenscar development report [15] and consists of a simple stopwatch that counts and displays seconds. The watch has three buttons: the first starts the timer, the second stops it and the third resets the counter to zero but does not change whether the watch is running or not.

The program comprises three main packages: a User interface, a Timer task and a Display manager. A fourth package provides a set of constants that can be used to "tune" the behavior of the program. This package will not be modeled because it contains only specification parameters for other packages, as happens with specifications packages, but it's data will be used in the translating process.



```
procedure StartClock
is
begin
   Ada.Synchronous_Task_Control.Set_True (Operate);
end StartClock;

procedure StopClock
is
begin
   Ada.Synchronous_Task_Control.Set_False (Operate);
end StopClock;
```

```
with Timer, Display;
package body User
is
   Buttons : PT;

   protected body PT is
      procedure StartClock
      is
      begin
         Timer.StartClock;
      end StartClock;

      procedure StopClock
      is
      begin
         Timer.StopClock;
      end StopClock;

      procedure ResetClock
      is
      begin
         Display.Initialize;
      end ResetClock;
   end PT;
end User;
```

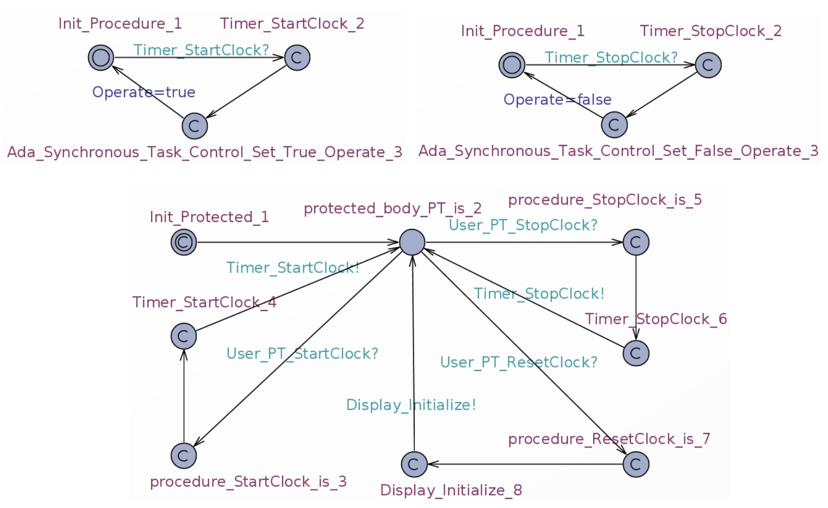
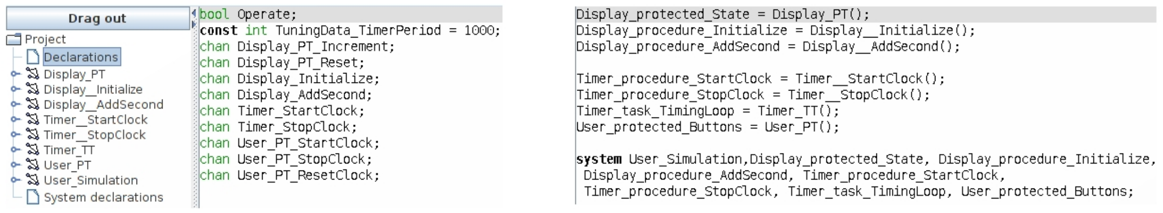


Figure 6: Stopwatch User package and the procedures associated with it.

The user package contains a protected type with an interrupt handler for each of the three buttons. The start and stop button alter a suspension object that controls whether the timer task runs. The reset button calls the display manager to zero the second count. The timer package, described in figure 4, declares the suspension object Operate and procedures to allow it to be set and reset. It also declares a periodic task that actually counts the seconds timed by the stopwatch.

Finally, the Display package, described in figure 3, maintains an internal counter for the seconds and also protects an output port which causes the counter to be displayed on the stopwatch's screen.

The following figure shows how the uppaal system declarations will look like after the program is fully translated.



```
bool Operate;
const int TuningData_TimerPeriod = 1000;
chan Display_PT_Increment;
chan Display_PT_Reset;
chan Display_Initialize;
chan Display_AddSecond;
chan Timer_StartClock;
chan Timer_StopClock;
chan User_PT_StartClock;
chan User_PT_StopClock;
chan User_PT_ResetClock;
```

```
Display_protected_State = Display_PT();
Display_procedure_Initialize = Display__Initialize();
Display_procedure_AddSecond = Display__AddSecond();

Timer_procedure_StartClock = Timer__StartClock();
Timer_procedure_StopClock = Timer__StopClock();
Timer_task_TimingLoop = Timer_TT();
User_protected_Buttons = User_PT();

system User_Simulation,Display_protected_State, Display_procedure_Initialize,
 Display_procedure_AddSecond, Timer_procedure_StartClock,
 Timer_procedure_StopClock, Timer_task_TimingLoop, User_protected_Buttons;
```

Figure 7: Stopwatch model global declarations and system declarations.

Program verification. After the translation process of a program we now have a number of interactive simulation and verification possibilities with which we can test the correctness of the source code. For example, our stopwatch program consists only in the stopwatch's functionality. In order to simulate it's interaction with the user another timed automata can be added to the system to model the user's behavior. With this automata we add user

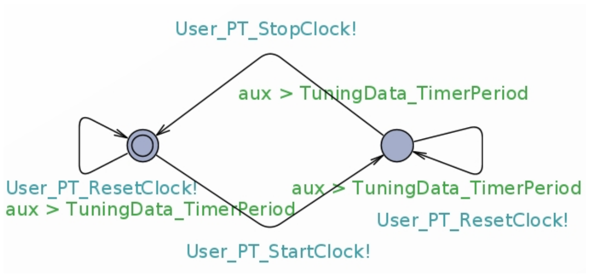

User_PT_StopClock!
aux > TuningData_TimerPeriod
User_PT_ResetClock!
aux > TuningData_TimerPeriod
aux > TuningData_TimerPeriod
User_PT_ResetClock!
aux > TuningData_TimerPeriod
User_PT_StartClock!

Figure 8: User simulation automata.

functionality by activating the button signals. Timing restraints state only that the user will wait at least the value of *TunningData_TimingPeriod*, which in our system corresponds to one second, until another stopwatch button is pushed. This easily allows the programmer to use a different number of user behaviors through automata specification and test the program's robustness.

In a more formal way, we can test the program's timing constraints with the aid of temporal logic rules: Uppaal offers simulation and verification functionality based on the model checking of a subset of TCTL logic [4]. Timing requirements (target properties to be checked) can be specified using the editing facilities of the GUI, or separately in a file. The following properties have been proven correct in our stopwatch uppaal system.

```
A[]  not  deadlock
A[]  Timer_task_TimingLoop.Release_Time <= 1000
```

Through the use of counter-examples, uppaal's model-checking engine proves the failure of any property and allows the programmer to detect errors in his code and correct them.

5 Conclusion and Future Work

Various different programs have been used to test the tool's translating capabilities, such as the Minepump program, an Autopilot simulation program, among others. As can be expected, each new program showed a number of new Ada instructions which, associated with different coding styles, brought new difficulties and challenges to the translation process.

The main goal of this development stage has been to enlarge the processing scope of the tool in order for it to encompass a reasonable section of Ada's sequential instructions and generate a secure and solid translation algorithm. This has been partially achieved with results the have become trustworthy with continued testing and development. However, to extract formal proofs of the system's correction based on this syntactic translation further work must be yet performed.

Another dificulty encountered consists in the size of the program: Larger program's increase the complexity of the translation process. Real-time systems are often composed by many modules, and when processing communication aspects in large programs, several difficulties arise. In order to accurately simulate the program's behavior the concurrency model has to be well taken into account. For instance, when modeling 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 that must be addressed in the near future.

Future work scheduled for the tool's development will adress the completion of the algorithm that automatically infers properties using a larger set of code annotations. 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 analyze 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] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model checking.* Cambridge, MA, USA: MIT Press, 1999.

[2] 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.

[3] ACM, "Association of computing machinery (acm) special interest group on ada (sigada) asis home page," 2009.

[4] 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.

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

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

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

[8] T. A. X. Project, "Xalan," Nov. 2007.

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

[10] 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.

[11] P. N. Amey and B. J. Dobbing, "Static analysis of ravenscar programs," *Ada Lett.*, vol. XXIII, no. 4, pp. 58–64, 2003.

[12] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada.* Cambridge University Press, 2007.

[13] 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.

[14] 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.

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

[16] 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*, no. 3185 in LNCS, pp. 200–236, September 2004.