An Approach to Model Checking Ada Programs[⋆]

José Miguel Faria[1,2], João Martins[1], and Jorge Sousa Pinto[1]

[1] Departamento de Informática / CCTC, Universidade do Minho, Braga, Portugal
[2] Critical Software, SA

Abstract. This paper describes a tool-supported method for the formal verification of Ada programs. It presents ATOS, a tool that automatically extracts a model in SPIN from an Ada Program, together with a set of properties that state the correctness of the model. ATOS is also capable of extracting properties from user-provided annotations in Ada programs, inspired by the Spark Annotation language. The goal of ATOS is to help in the verification of sequential and concurrent Ada programs based on model checking. The paper introduces the details of the proposed mechanisms, as well as the results of experimental validation, through a case study.

1 Introduction

Ada [18] is a programming language highly recommended for the development of critical software systems, due to its careful and safe design, and the existence of clear guidelines for building this kind of systems. Given that high integrity systems failures may have severe consequences, the use of formal verification techniques can be of high value. The tool presented in this paper enhances the relationship between formal methods and critical systems, enabling model checking of Ada programs.

Formal methods encompass several verification techniques such as deductive verification [12,13], model checking [16], and theorem proving [7]. Model checking is notably one of the most successful and is at the core of the solution presented in this paper. Given a model of a system and a property to verify, model checking answers yes or no to the question "Does the model satisfy the property?".

The application of the model checking technique to software is seen as very promising and had led to the creation of a new research area, designated software model checking [8,15]. This technique is not simply the application of model checking to software, it involves the resolution of some obstacles in its application, like state explosion or model construction problems (see Section 1.1). The main goal of the ATOS tool is to help overcoming some of these obstacles in the application of model checking to sequential and concurrent Ada programs, with a special focus on the concurrent ones.

[⋆] This work is partially supported by the EVOLVE Project, "Iniciativa QREN, SI&IDT 1621", funded by "Fundo Europeu de Desenvolvimento Regional", through "Programa Operacional". J. M. Faria was also supported by FCT (the Portuguese Foundation for Science and Technology) PhD grant `SFRH/BDE/51049/2010`.

Software model checking tools follow essentially two main approaches: (i) either generate input models for one or more different existing model checkers, where ATOS is included, or (ii) include their own model checking algorithms, to check the correctness of the models, like BLAST [5] or SLAM [1]. ATOS uses the SPIN model checker [14] to check the correctness of the translated models. SPIN is a model checking tool focused on verifying the correctness of concurrent systems models which clearly matches our intents. The models are described in PROMELA, the SPIN modeling language, and correctness claims can be stated through Linear Temporal Logic (LTL) [17] formulas or PROMELA assertions.

ATOS takes an Ada program and extracts from it a PROMELA model, which simulates the runtime behavior of the first. The correctness of the generated models, and consequently of the Ada program, is stated through a set of desire properties in the model. These can be automatically inferred by ATOS or specified in the Ada program using an annotation language inspired by SPARK [2].

1.1 Software Model Checking Challenges

The application of model checking in the verification of software systems poses several challenges. Most notably:

Model construction: Manual model construction of software systems is an error prone process and time consuming due to the complexity of these systems. In addition, there is a gap between the semantics of programming languages for software systems (e.g. C, Ada, or Java) and the input languages of model checking tools. Programming languages have richer features with more complex semantics than modeling languages.

State explosion: This is recognizably the biggest problem of model checking. In software model checking the problem can become even more serious, due to the size of software systems, which leads to the generation of models with a lot of states. Thus, more aggressive abstractions must be considered.

Property specification: Typically, properties are specified in some variant of temporal logic. This encompasses two difficulties: Firstly, it requires some level of expertise for expressing the desired system properties in temporal logic. Second, the mapping of these properties to the properties of the model may not be straightforward, since the typical specification languages are designed to state properties of mathematical models rather than source code properties.

Output interpretation: When a property does not hold in a given model, the model checker reports a counter-example illustrating a trace that evidences the violation of the property. Large models can produce very long traces. As such, manually matching up the provided trace with the model's source code can be really a hard job.

ATOS directly addresses the first and third identified challenges: given an Ada program, it is capable of automatically extracting a PROMELA model from it, and also inferring properties directly from the program. ATOS is an ongoing project; at this stage the remaining two challenges have still not been considered. However, by converting Ada programs into models of such a powerful model checking tool like SPIN, ATOS profits from all of its abstraction techniques, which can be a great help in dealing with the state explosion problem.

2 Running Example

The case-study is a solution to the **Readers-Writers** well known concurrency computing problem. It is extracted from [3] and covers some of the Ada features that ATOS can translate. The Ada program is constituted by the main `Readers_Writers` procedure that contains the declaration of a single protected object `RW`, a single `Writer` task, and a task type `Reader`, which is instantiated twice. The main procedure has only one statement (mandatory), which is the *null* statement. The details of the other components will be discussed next.

2.1 The RW Protected Object

The protected object `RW` is composed by four protected operations and two data structures declared in the private part. Its declaration is shown in list 1.1.

Listing 1.1. Protected Object

```
Protected RW is
    procedure EndRead;
    procedure EndWrite;
    entry StartRead;
    entry StartWrite;
 private
    Readers:natural range 0..2:=0;
    Writing:boolean :=false;
 end RW;

Protected body RW is
    procedure EndRead is
      begin
        Readers:=Readers - 1;
    end EndRead;

   procedure EndWrite is
      begin
        Writing:=false;
   end EndWrite;

    entry StartWrite when not Writing and Readers = 0 is
      begin
        Writing:=true;
    end StartWrite;

    entry StartRead when not Writing is
      begin
       Readers:=Readers + 1;
    end StartRead;
end RW;
```

The variable `Readers` contains the number of readers along the program execution, while the boolean `Writing` signals whether there is a writer writing. The values of these two variables are altered by four protected operations, two entries (`StartWrite` and `StartRead`) and two procedures (`EndWrite` and `EndRead`). The entries have a barrier, which in the case of `StartWrite` means that in order for a writer to start writing there can be no readers reading and no writers writing, while in the case of `StartRead` it indicate that there can be no writers writing in order for readers to start reading.

2.2 The Writer and Reader Tasks

The `Writer` task is a single task, while the `Reader` task is a task type, which means that several instantiations of it may be created during the program execution. In our example, as mentioned, it is instantiated twice. Lists 1.2 and 1.3 correspond to the bodies of `Reader` and `Writer` tasks, respectively. The behavior defined by these two body tasks is similar: it is a loop trying to start and end the read/write of the protected object.

Listing 1.2. The body of task Reader

```
task body Reader is
    begin
        loop
          RW.StartRead;
          RW.EndRead;
        end loop;
    end Reader;
```

Listing 1.3. The body of Task Writer

```
task body Writer is
  begin
    loop
       RW.StartWrite;
       RW.EndWrite;
    end loop;
end Writer;
```

3 Model Extraction

The ATOS tool is capable of translating a subset of the Ada language into PROMELA models. All the syntax and semantic information of Ada Programs required for the translation is provided by Ada Semantic Interface Specifications (ASIS) [6], a library that offers an excellent interface to the Ada syntax tree programs (AST). ATOS handles the following Ada declarations: subprograms (procedures and functions), packages, concurrency primitives (tasks, protected objects, and entries), variable declarations (including arrays and basic records), integer constants, and new integer types and subtypes. The translation of many statements as well as logical operators conversion or the support for the Ada inheritance mechanism are also provided by ATOS. Next, the translation details for some of these primitives are given.

3.1 Encapsulation

Encapsulation is a well known mechanism used by most of programming languages, including Ada, which restricts and hides object's data. This mechanism is not directly matched with PROMELA semantics; nevertheless, it is partially assured by the ATOS translation. The encapsulation is guaranteed for variables declarations; all other declarations must have different names in order to avoid the redefinition of these along the extracted model.

ATOS preserves the encapsulation of variables declaring them globally in the model, prefixed with the name of its "mother entity". For example, the variable `Readers` from the protected object `RW` is declared globally as `RW_Readers`. The variable renaming process is performed automatically by ATOS. This solution was reached after finding difficulties in the use of PROMELA primitives scopes, namely due to the fact that PROMELA **processes** (tasks) do not allow the declaration of **inlines** inside its "body", which are used, for example, to translate procedures and functions. This restriction forced inlines primitives to be declared globally (i.e. not encapsulated) as well as its variables, so that tasks can access them.

This approach allows the use of local variables in the specification of global properties (see section 4). However, for this case, the user must provide the prefix of the variable with the name of the mother entity, so that ATOS can know which variable he is referring to.

3.2 Subprograms

Subprograms encompass functions and procedures whose execution can be invoked through a procedure call. Subprograms parameters have a mode (**in**, **out**, or **in out**), and they can be passed either by copy or by reference. A parameter passed by copy denotes a separate object from the actual parameter and the information transfered between the two happens only in two moments: immediately before and after the subprogram execution. To simulate this, ATOS creates an auxiliary variable for each, which is assigned with the value of parameter before the beginning of the procedure execution and all occurrences of this parameter inside the procedure are replaced by the corresponding auxiliary variable. At end of the procedure execution, the parameter is assigned with the corresponding auxiliary variable. A parameter passed by reference is updated along the subprogram execution.

There is no direct equivalent to an Ada procedure in PROMELA; ATOS translates procedures as **inlines**, a primitive which simply defines a replacement code for a designated name possibly with parameters. The procedure statements are converted into inline statements, whereas the declarations are made global as explained in Section 3.1. The procedure parameters are translated into inline parameters. However, unlike parameters of subprograms, they do not have a type and a mode associated, they are simple names. This does not become a problem since ATOS only converts an Ada program if its compilation was successfully, therefore the parameters type and mode checking are guaranteed up-front.

3.3 Concurrency

The concurrency model of Ada is based on three main primitives: tasks, protected objects, and shared data. The communication between concurrent primitives follows the rendezvous mechanism, which is reproduced in the extracted models.

Concurrent Ada programs may be executed on a single processor (interleavead) or on multiprocessors. The concurrency model of SPIN is different from Ada's, the behavior of SPIN models is defined simply by the arbitrarily interleaving of the processes statements.

Tasks: The definition of an Ada task is divided in two parts: the specification, which describes the interface with other tasks, and a body that contains the code defining the task's behavior. A task can either be declared as a single task or a type task. The first becomes active from the moment it is declared; whereas the type task simply creates a new type which can be instantiated later.

A task of an Ada program is translated into a PROMELA process. Similarly to Ada tasks, processes in PROMELA are the only primitive that can represent parallel activities. Single tasks are converted into **active proctypes**, which become active from the moment they are declared. The type tasks are translated into PROMELA **proctypes**, which simply creates a new process type that can be instantiated later. However, ATOS creates a new type for each instantiation due to encapsulation issues: Because all the variables from an Ada program are declared globally, the variables of a task type would be the same for any number of instantiations because they are not encapsulated. To overcome this, ATOS creates a new type and new declarations for each instantiation. Lists 1.4 and 1.5 illustrate the conversion of the task type `Reader`.

Listing 1.4. First instantiation of task type `Reader`

```
proctype Reader0(){
 do
   :: StartRead ( );
      EndRead ( );
 od  }
```

Listing 1.5. Second instantiation of task type `Reader`

```
proctype Reader1(){
 do
   :: StartRead ( );
      EndRead ( );
 od  }
```

A process has associated an ID number which univocally identifies its sent messages. Processes have also an associated channel through which it can receive messages from other processes.

Protected Objects: They are a structured mechanism which provide mutually-exclusive access to shared data. A protected object is relatively similar to a package, the main difference is the fact that all operations of a protected object are mutually-exclusive. The operations encompass three different declarations: procedures, functions, and entries.

Similarly to tasks, protected objects can be either declared as single or type, and are translated to PROMELA processes, i.e, **active proctype** or **proctype**, respectively. Associated to the process, ATOS also creates two PROMELA channels, one for queuing the blocked entries and another for communicating with other processes, in order to ensure the correct simulation of Ada protected objects. Only one operation is allowed to start at each time, except functions which can execute more than one simultaneously. If the operation is an entry, it must evaluate the barrier expression before executing: If the barrier is open the operation is executed; otherwise the process (an Ada task) which is trying to execute the operation communicates to the corresponding protected object process that the barrier is closed and is enqueued. Every time a process finishes the execution of a protected operation, it communicates this to the protected object process, which then activates the enqueued processes to test again the entries barriers before opening the semaphore again.

The procedures and functions operations are translated equally to other procedures and functions. The only difference is that, in the beginning of the operation, there is a small piece of code that tries to acquire permission to execute. The translation of entries is similar to the conversion of procedures. Entries are converted as inlines and its parameters as inline parameters. However, they have a few extra statements which test the barrier expression and communicate the result to the protected object, in order for this to know if it is closed or not.

3.4 Types and Variable Declarations

The Ada and PROMELA type systems are very different, the first one offers a much more powerful and wide type system. Table 1 illustrates the correspondence between Ada predefined types and the PROMELA types defined by ATOS.

The range of values that PROMELA types can represent is always greater or equal to the corresponding Ada types ranges. As such, when corresponding types have different ranges, overflow errors could stay undetected in the model. For example, a variable with type `short` in PROMELA can represent the number -2^{15} which it is not possible with correspondent Ada type. This is easily avoided adding an LTL formula which asserts that a variable respects its range values.

Type Ada	Range Ada	Type PROMELA	Range PROMELA
boolean	false, true	bool	false, true
integer	$-2^{15}+1..2^{15}-1$	short	$-2^{15}..2^{15}-1$
positive	$1..2^{15}-1$	unsigned	$0..2^{n=15}-1$
natural	$0..2^{15}-1$	unsigned	$0..2^{n=15}-1$

Table 1. Correspondence between Ada predefined types and PROMELA types

ATOS is capable of converting variables with the Ada predefined types from Table 1 and with new integer types and subtypes defined in an Ada program. ATOS also allows the use of range constraints in variables declarations and takes advantage from this range to reduce the size of variables. For example, the natural variable `Readers` with range 0..2 is declared as an unsigned with the smallest number of bits that can represent the upper bound of range (2 bits in this case), instead of being declared as an unsigned with 15 bits.

Arrays: An array is a data structure which aggregates a list of elements, all of the same type. This structure exist both in Ada and PROMELA. However, in Ada it is a more powerful data structure, since it allows the declaration of complex expressions which define the range of an array. In PROMELA, the range of an array can only be defined within $0..\texttt{N}-1$, where $\texttt{N} \in \mathbb{N}$ and represents the number of elements. In Ada, the range of an array is defined within `N..M`, where $\texttt{N},\texttt{M} \in \mathbb{Z}$ and $\texttt{N} \leq \texttt{M}$, thus allowing the definition of the lower and the upper bound of an array range unlike in PROMELA arrays, where just the upper bound is defined.

ATOS can convert all declarations of arrays in Ada, except those which have a predefined range such as: **Character** or **Positive**. In the conversion of an Ada array to a PROMELA array, ATOS firstly calculates the number of elements defined in Ada's array, which is given by this formula:

$$Nr\_Of\_Arrays\_Elements = Upper\_Bound - Lower\_Bound + 1,$$

where `Upper_Bound` and `Lower_Bound` are the upper and lower bound, respectively, of an Ada array range. With the number of elements ATOS can declare the array in PROMELA, but the translation work it is not over, because PROMELA arrays can only be accessed from indexes between $0..\texttt{Nr\_Elements}-1$. This problem is solved calculating the difference between 0 (the lower bound of a PROMELA array) and the lower bound of an Ada array; the calculated offset is then added to the indexes value of a PROMELA array every time it is accessed.

3.5 Statements

Statements are a familiar concept to most of programming languages, including Ada. This concept also known as instruction exists in PROMELA too. ATOS is able to convert the following Ada statements: if, null, assignment, case, loop, exit, while loop, for loop, goto, procedure call, accept, selective accept, return, entry call, and block. As an example, the details of translating an **accept** statement are given here.

The accept statements are declared in the specification part of a task (as an entry declaration) and identify the interaction points of a task. The translation of this statement is done by just mapping its semantic into PROMELA models. In PROMELA, processes (tasks) communicate with each other through channels, so an accept statement is mapped simply as an execution point where a task is listening on a channel that will eventually receive a message from other task, where the message contains the sender identification and the parameters of the accept statement. The general PROMELA code for the translation of an accept statement is given in the below List.

```
*AcceptName* ? *SenderID*,*Parameters* ->
                    *Statements*
                    Processes[*SenderID*] ! *AcceptName*
```

A sender task remains suspended until the requested task finishes the accept statement execution. When the requested task reaches the end of an accept statement, it sends a message to the sender task in order to activate it again, as it happens in Ada.

3.6 Main Program

The concept **main** program exists in Ada, despite not being identified with a special name as it happens in other programming languages (e.g. Java or C). The main of an Ada program is translated into a PROMELA `init`, which is an parameterless active process that can be declared only once per model. The `init` process contains the correspondent Ada main program statements and possibly instantiations of task types. The processes instantiations in PROMELA are performed inside other processes because they are executed through the `run` statement, rather than by a declaration as in Ada. The procedure **Readers-Writers** is the main subprogram of the running example (see Section 2) and is converted to PROMELA as shown in the below List.

```
init{ run Reader0();
      run Reader1();
      //statements
      skip;  }
```

4 Properties Specification

A model checker verifies whether a model fulfills a given (set of) property(ies). Hence, the specification of properties is a crucial step in the process. ATOS offers high level mechanisms for the verification of extracted models, based on (SPARK inspired) annotations at source code and on automatic inference of properties from Ada programs. To exemplify the mechanisms supported by ATOS, we start by enumerating a set of expected requirements (properties) for the **Readers-Writers** example:

1. The numbers of readers lies between 0 and 2 (the max number of readers).
2. Before a reader finishes to read there must be at least one reader reading.
3. After a reader ends to read, the number of readers is decreased in one.
4. Readers and writers can not execute simultaneously.
5. A writer does not change it status (writing or not writing) when a reader finishes reading.
6. The system is deadlock free.

Range Checking: This mechanism extracts an LTL formula for each variable declared with a range constraint, which checks whether the range constraint is violated or not. The requirement 1 can be translated at code level, checking if the variable `Readers` does not violate its range constraint. The specification of this property is done automatically by ATOS, which generates the following LTL formula:

$$ltl\ RC0\ \{\ [\,](RW\_Readers >= 0\ \&\&\ RW\_Readers <= 2)\}$$

Temporal properties: The specification of temporal properties in ATOS is restricted by the temporal logic allowed in SPIN, which is the LTL. ATOS offers a high level mechanism for the specification of temporal properties, based on the properties pattern for LTL defined in [9]. These are composed of five basic patterns: *universal*, *absence*, *response*, *existence* and *precedence*. The five patterns have variations which are defined in terms of five basic pattern scopes:

– a pattern holds *globally* along the program execution;
– a pattern holds *after* the first execution of a specified event;
– a pattern holds *before* the first execution of a specified event;
– a pattern holds *between* the occurrence of a designated event and the occurrence of another specified event;
– a pattern holds *after* the occurrence of a specified event and *until* the next occurrence of another event, or throughout the rest of program execution if there is no further occurrence of that event before the end of program.

Requirement 4 of the current example can be specified using these patterns:
- -# property $RW\_Readers > 0$ **and** $RW\_Writing$ **is_false globally**

This annotation is then converted by ATOS into the LTL formula:

$$ltl\ prop0\ \{[\ ]\ (\ !(RW\_Readers > 0\ \&\&\ RW\_Writing)\ )\}$$

In addition to these set of patterns, the user can yet specify its own temporal properties. By default, ATOS adds automatically the property stating deadlock freedom (requirement 6).

Asserts, Preconditions and Postconditions: These annotations allow for the verification of conditions at a certain point during a program execution and are converted into PROMELA asserts. An annotation corresponding to an assert can be specified anywhere in an Ada program where statements are allowed. Pre- and postcondition annotations are defined only in the body of the followings Ada primitives: functions, procedures and entries. Precondition statements (assert) appear at the beginning of the corresponding primitives; postconditions appear at the end.

Requirement 2 and 3 can be expressed by a precondition and a postcondition annotation, respectively, in procedure `EndRead`. The precondition states that the value of `RW_Readers` must be greater than zero:

- -# pre $RW\_Readers > 0$,

while the post condition checks if the value of `RW_Readers` at the end of the procedure is equal to the value of `RW_Readers` at the beginning of procedure (old value) less one:

- -# post $RW\_Readers = RW\_Readers \sim -1$

In order to simulate the old value of a variable, ATOS creates an auxiliary variable which is assigned with the value of the corresponding variable at the beginning of the Ada primitive.

Invariants: This mechanism allows for the specifications of properties for checking whether a given logic expression is valid along the execution of one of these Ada primitives: procedure, functions and entries. An invariant annotation is given in the body of the enunciated Ada primitives like preconditions and postconditions.

Requirement 5 can be verified stating that the boolean `RW_Writing` is not altered along the procedure `EndRead` execution:

- -# invariant $RW\_Writing = RW\_Writing\sim$

ATOS converts the invariant annotations in several LTL formulas, one for each task that possibly executes the Ada primitive. In this example, there are two processes (tasks) which can possibly execute the procedure `EndRead` (the two instantiations of `Reader` task), so ATOS generates two LTL formulas equivalent to this pattern annotation:

$RW\_Writing = RW\_Writing\sim$ **is_true between** Q **andOp** R

where Q/R corresponds to the states whereupon the `Reader` processes begin/end the `EndRead` execution. The invariant is verified if the two LTL formulas are valid.

5 Related Work

Software model checking is a wide research area. Several projects have been developed with the intention of applying model checking techniques to software. Among these, some are directly related with Ada programs, namely Quasar [11] and Ada Translating Toolset [10]. The first also extracts models directly from Ada programs, but it uses Colored Petri Nets to represent the extracted models. The second one uses an intermediate representation of Ada programs before it generates the correspondent models.

The approach followed by ATOS and Quasar allows the extraction of more accurate models comparing to Ada Translating Toolset because some program details are lost due to the intermediate representation in this last. However, the Ada Translating Toolset has the capability to extract more easily models for different model checkers because its intermediate representation is closer to a model than Ada programs.

6 Conclusions and Future Work

The tool presented in this paper provides mechanisms to extract models and property specifications from Ada programs to the SPIN model checker. Although the use of a modeling language to represent programming language features imposes some natural restrictions, the model extraction covers a wide variety of Ada features and generates models that closely relate to the correspondent Ada programs. Yet, there is still a gap in the simulation of Ada concurrent programs which is the absence of a scheduler implemented in SPIN that would approximate even more the behavior of Ada concurrent programs to the extracted models.

A solution to solve this gap and the development of mechanisms which address other software model checking challenges (state explosion and output interpretation) remains as future work. ATOS has been tested with several concurrent small/medium case studies, like readers-writers or producers-consumers, and it is currently being tested with a bigger case study.

References

1. Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *In: IFM. (2004*, pages 1–20. Springer, 2004.
2. John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
3. M. Ben-Ari. *Principles of Concurrent and Distributed Programming (2nd Edition) (Prentice-Hall International Series in Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
4. Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer, 1 edition, 2008.
5. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9:505–525, October 2007.
6. James B. Bladen, David Spenhoff, and Steven J. Blake. Ada semantic interface specification (asis). In *Proceedings of the conference on TRI-Ada '91: today's accomplishments; tomorrow's expectations*, TRI-Ada '91, pages 6–15, New York, NY, USA, 1991. ACM.
7. Robert S. Boyer and J Strother Moore. Proof-checking, theorem-proving and program verification. Technical report, 1983.
8. Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.
9. Matthew Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, 1998.
10. Matthew B. Dwyer, Corina S. Pasareanu, and James C. Corbett. Translating ada programs for model checking : A tutorial. Technical report, 1998.
11. S. Evangelista, C. Kaiser, J. F. Pradat-Peyre, and P. Rousseau. Verifying linear time temporal logic properties of concurrent ada programs with quasar. *Ada Lett.*, XXIV:17–24, December 2003.
12. Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
14. Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.
15. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, October 2009.
16. Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

17. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
18. S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal 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)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.