INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL
ISSN 1841-9836, 11(5):645-656, October 2016.

Statistical Automaton for Verifying Temporal Properties and Computing Information on Traces

A. Ferlin, V. Wiels, P. Bon

Antoine Ferlin*
1. IFSTTAR, 20 rue Élisée Reclus,
BP 70317, 59666 Villeneuve d'Ascq Cedex, France
2. ONERA/DTIM, 2 avenue E. Belin
BP74025, 31055 Toulouse, France
3. AIRBUS OPERATIONS S.A.S.,
316 route de Bayonne, 31060 Toulouse Cedex 09, France
*Corresponding author: antoine.ferlin@ifsttar.fr

Virginie Wiels
ONERA/DTIM, 2 avenue E. Belin
BP74025, 31055 Toulouse, France
virginie.wiels@onera.fr

Philippe Bon
IFSTTAR, 20 rue Élisée Reclus,
BP 70317, 59666 Villeneuve d'Ascq Cedex, France
philippe.bon@ifsttar.fr

Abstract: Verification is decisive for embedded software. The goal of this work is to verify temporal properties on industrial applications, with the help of formal dynamic analysis. The approach presented in this paper is composed of three steps: formalization of temporal properties using an adequate language, generation of execution traces from a given property and verification of this property on execution traces. This paper focuses on the verification step. Use of a new kind of Büchi automaton has been proposed to provide an efficient verification taking into account the industrial needs and constraints. A prototype has been developed and used to carry out experiments on different anonymous real industrial applications.
Keywords: Statistical Büchi Automaton, Information Computation, Runtime Verification, Dynamic analysis, Linear Temporal Logic.

1 Introduction

Development of critical software is constrained by certification standards. More precisely, DO-178 concerns avionics software. It defines objectives for each step of the software development process and, in particular, for the verification phase. AIRBUS has investigated the use of several kinds of verification methods. These methods can be classified using two criteria: formal or not, static or dynamic. Classic verification means are typically not formal: test is dynamic, review is static. Formal techniques are typically static, but there has been a lot of work recently on run-time verification which can be classified as formal and dynamic.

The origin of this work is the verification of temporal properties in an industrial context. These properties cannot easily be verified using the current AIRBUS verification framework based on static analysis [18]. Another way consists in using dynamic analysis to perform the verification of temporal properties. Testing can be difficult for properties involving timing aspects. Simulation, which consists in testing a software on a simulated hardware, has been experimented

and allows the analysis of the program at each step of its execution. However, even if commandability and observations are easier with simulation than with testing, considered executions are very long and manual verification of a property is difficult and costly.

This paper presents a dynamic analysis approach to formally verify temporal properties on execution traces generated by simulation. The context of the industrial process restricts the dynamic analysis approach to off-line analysis techniques, i.e. a posteriori verification on execution traces. Constraints also exist on the generation of traces: values of variables at given instants are obtained by positioning explicit observation points in the program, and the number of such observation points must be minimized for efficiency reasons. Execution traces are generated using existing test cases. Test strategy is not considered in this paper. It is focused on the use of patterns of temporal properties to compute statistical information. The aim is double. The first goal is to conciliate the temporal properties which classically have a semantics on infinite traces, with finite traces. Secondly, when a property is violated, this method can provide additional information to the client instead of the basic "Ok/Failure" response. When bugs occurs, this method limits the necessary investigations, hence reduces the debugging cost.

An overview of the proposed approach is detailed in section 2. Section 3 positions our work according to related work. Section 4 recalls classic definitions and details the verification phase using automata that compute statistical information. Section 5 provides experiment results in terms of verification time and statistical information use. Finally, we conclude about our work and provide future work elements in Conclusions.

2 Overview of the approach

We are working within the Airbus simulation framework. This framework simulates hardware of the program to be verified. The simulated hardware is instrumented, so the program executed on this framework is the embedded software. Extraction of execution data consists in generating a trace with the help of observation points. They can only be defined using a debuggin interface. Our goal is to verify a temporal property on an execution trace of a given program. The first step consists in defining an adapted language to formalize a temporal property. The second step is trace generation. Verification of the property on the trace is the last step (Figure 1). In this paper, we focus on the last step. However, we give some essential elements of the first two steps hereafter.

The first step consists in identifying properties to be verified. Currently, the industrial methodology verifies temporal properties using non-formal static analysis. In other words, the verification phase is done by engineers using code review. Hence, the goal of this approach is typically focused on industrial needs. Actually, verification cost and time have to be reduced as much as possible and safety has to be at least identical with the present verification approach. In addition, the new equipped approach must be as simple as possible for operators.

Formalisation of temporal properties consists of four steps. Firstly, some properties are gathered from critical software. Secondly, properties are classified. Then, a dedicated language is defined from gathered classified properties. This language is based on a combination between a subset of LTL and regular expressions. Some operators on variables had been defined in order to have access to some information such as the time of the last modification of a variable. The language definition is not the purpose of this article and can be found in [9]. Use of a dedicated language is a consequence of the desire for simplification for operators. Finally, the properties studied are formalized using our dedicated language.

One issue with a posteriori dynamic analysis is gathering traces. Indeed, trace size quickly increases with execution time and with the number of collected variables. We use two ways to minimize traces. Firstly, we only collect, at each step of the program (a step is a C instruction),

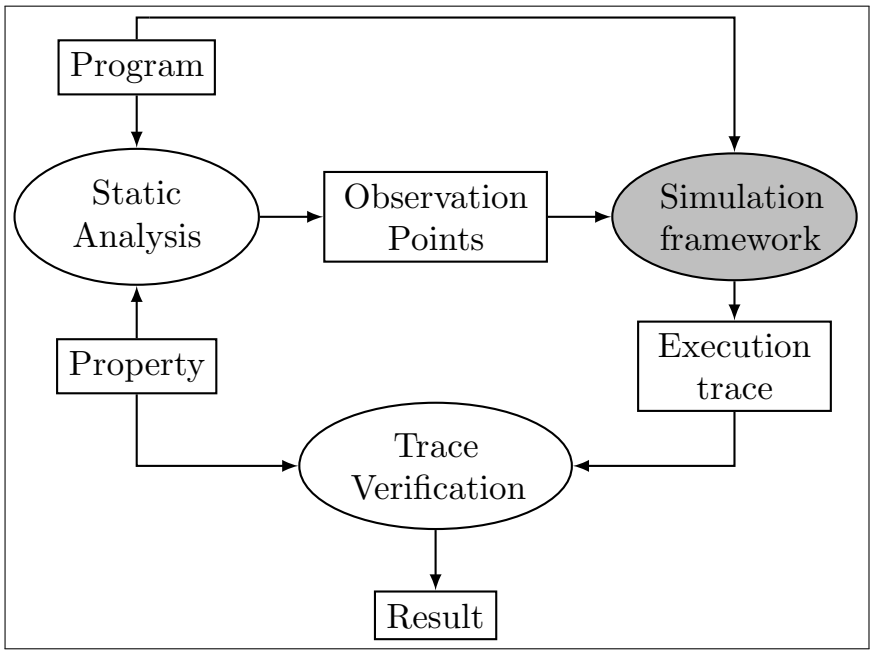

Figure 1: Overview of the approach

variables which are necessary to verify a given property. Secondly, we only collect these variables if their value changes. More details on trace generation can be found in [8].

This last step consists in verifying a formalized property on an execution trace. This step needs as input an execution trace given by the previous steps. Properties are formalized using language defined in the first steps and are then transformed into a Büchi automaton which is executed on the trace to be verified.

3 Related Work

Existing work on run-time verification [1] can be classified in two categories: on-line or a posteriori. An "on-line" approach means that the verification is done during the execution of the programme, whereas an "a posteriori" approach is done on execution traces. On-line verification consists either in adding assertions to the programme to express the properties to be monitored during the execution, or in executing, parallel to the programme, an automaton representing the property and taking as inputs data from the programme execution. A lot of work exists for on-line verification, especially for Java programmes and in the aspect-oriented programming community [7,12,13,16,19]. We work on C programmes which are either sequential hard-realtime programmes, or multitask realtime programmes (ARINC 653, POSIX platforms) and for certification reasons, we are restricted to an "a posteriori" verification. Existing works are fewer and differentiated by the following criteria.

The first criterion is the nature of the considered execution traces. In several existing works, complete execution traces can be obtained by listening to all the variables of the programme [3, 17]. This is not the case in our context: traces are obtained by positioning observation points in given places of the programme and we use static analysis to compute a minimized number of observation points. Indeed, the bigger the number of observation points, the bigger the size of the execution trace.

The second issue is the implementation of an efficient verification technique on the traces obtained that can be very large. Existing works are based on rewriting techniques [12] or specific techniques, such as translation of LTL formula into state machines [11] [6]. We use an existing

tool for LTL properties and propose specific techniques for regular expressions and parametric properties.

The last criterion is the way to handle finite traces. Classic models in temporal logic are infinite, while we want to verify properties on finite traces. Existing work proposes different solutions to this problem. [2] proposes an adaptation of the temporal language semantics. [15] and [4] propose a multi-valued logic to handle the cases where it is not possible to conclude on the satisfaction of a formula. In particular, [4] needs to generate two Büchi automata (one for the LTL_3 property ϕ and one for $\neg\phi$). This method requires time when the property depth is high. [15] also gives an interesting account of different work dealing with the finite trace issue and especially work that explores means to decide whether all possible infinite completion of a finite trace verifies a given property. A simple solution can also be to loop on the last state in order to render a finite trace infinite, but it modifies the satisfiability of some formulae [13]. We propose a simple pragmatic approach to this problem in section 4.2.

4 Verification of properties on execution traces

The last step of our approach consists in verifying a formalised property on an execution trace. This step needs as input the formalised property and a corresponding trace. The property is formalized and the trace is generated according to the method proposed in 2. The temporal property is translated into a Büchi automaton using *Ltl2ba* [10]. Hence, it is executed on the trace to perform the evaluation of the mapping formula.

In order to explain our approach, let us recall the definition of a Büchi automaton, which is based on the definition of the classic automaton.

4.1 Definitions

[14] defines an automaton as follows:

Definition 1 (Automaton)**.** An automaton is a 5-uplet $A = (Q, \Sigma, \rightarrow, q_0, F$ such that:

- Q is a set of states
- Σ is an alphabet
- $\rightarrow \in Q \times \Sigma \times Q$ is a transition relation
- q_0 is an initial state
- $F \subseteq Q$ a set of final states.

Definition 2 (Accepting condition)**.** A word $w \in \Sigma^*$ n length sized is a word of $\mathcal{L}(A)$ [14] where A is an automaton, if and only if there is a sequence $(q)_{i,i\in[\![0,n]\!]}$ which begins at q_0, such that $\forall i \in [\![0, n-1]\!]$, $(q_i, w_i, q_{i+1}) \in \rightarrow$ and $q - n \in F$.

An automaton A recognises the regular language $\mathcal{L}(A)$ defined on the Σ alphabet. All words of $\mathcal{L}(A)$ are finites words.

The Büchi automaton definition can now be expressed using the automaton definition [5]:

Definition 3 (Büchi Automaton)**.** A Büchi automaton is an automaton such that the accepting condition of a word is modified, in order to accept infinite words.

Definition 4 (Accepting condition)**.** A word $w \in \Sigma^\omega$ is a word of $\mathcal{L}(B)$, where B is a Büchi automaton, if and only if there is a sequence $(q)_{i,i\in\mathbb{N}}$ which begins at q_0 and such that $\forall i \in [\![0, n-1]\!]$, $(q_i, w_i, q_{i+1}) \in \rightarrow$ and $\forall j \in \mathbb{N}, \exists k \in \mathbb{N}, k > j$ and $q_k \in F$.

After the recall of these definitions, the next step consists in handling finite trace with a Büchi automaton which only accepts infinite words or traces.

4.2 The finite trace problem

Let us assume that ϕ is a temporal property and σ_∞ an infinite trace. The property is transformed into a Büchi automaton $\mathcal{B}_\phi$. Hence, $\sigma_\infty \vDash \phi$ if and only if σ_∞ is recognized by $\mathcal{B}_\phi$.

We deal with finite trace, whereas the semantics of LTL is on infinite ones. A classic solution consists in transforming the finite trace into an infinite one by looping over the last element. But this solution introduces border effects which have to be controlled. Hence, if we have a result on the trace σ_∞, what will we able to conclude about the finite trace σ which derives from σ_∞? This is how we propose to respond to this question.

A specific algorithm has then been defined to handle the execution of the Büchi automaton at the end of the trace. This algorithm answers the following question: is there an infinitely often accessible final state? This algorithm performs:

1. Computing the strongly connected components of the Büchi automaton, by only taking into account transitions where the formula is true.
2. Defining a direct acyclic graph (dag), equivalent to the Büchi automaton, from the strongly connected components computation.
3. Browsing the direct acyclic graph for each element of the current state.
4. Determining for each state of the dag accessed from an element of the current state if it is a final state of the Büchi automaton.

We have thus implemented a verification algorithm for the property on the trace. But the transformation of a finite trace into an infinite one modifies the satisfiability of some formulae.

The satisfiability of a property may change according to the nature of a trace, as presented in the following instances.

Example 5. *The formula $\Diamond(\Box A \vee \Box\neg A)$ (which means: eventually we always have A or we always have not A) is true with a finite trace which loops at the end, because the last state is A or non-A (Figure 2,b)). With an infinite trace, this formula may be false (Figure 2,a)). For instance, if the trace alternates A and non-A at each state, the formula will be false.*

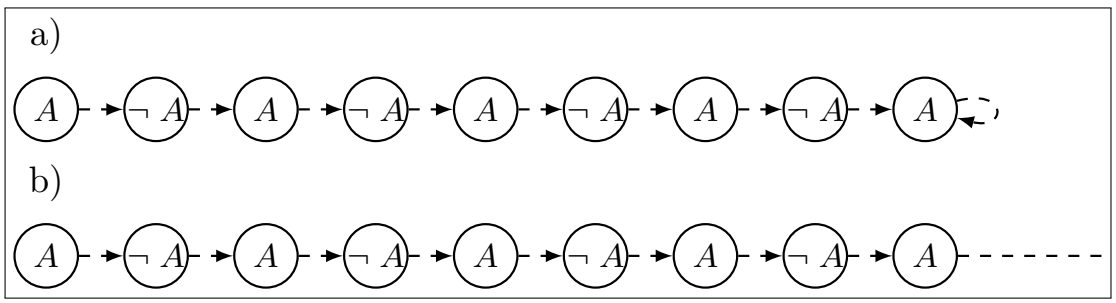

Figure 2: Two traces

In the considered context, the satisfaction of a formula may depend on the place where the execution is stopped in the program.

Example 6. *In the following example, the first trace is a theoretical trace corresponding to an infinite execution of the program. The other two traces are prefixes of the infinite trace where execution has been stopped at different execution times. The property $\Box(P \Rightarrow \Diamond Q)$ is false on the first prefix, but the same property is true on the second prefix.*

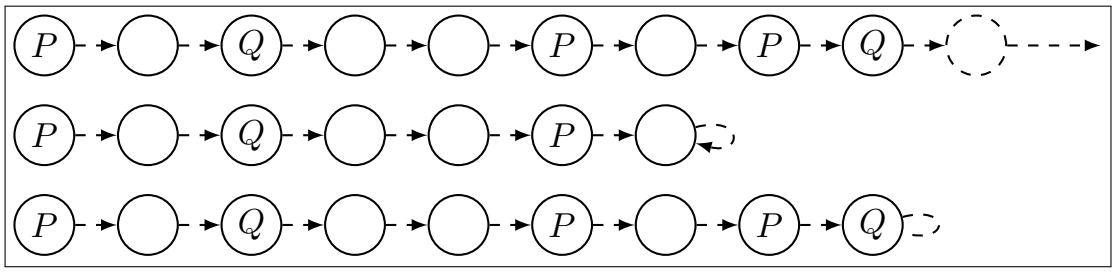

Figure 3: Satisfiability depends on the end of trace

In order to evaluate the interpretation of a property, we propose the following algorithmic method. ϕ is a property to be verified on an execution trace σ. $\mathcal{A}_\phi$ refers to the Büchi automaton associated to ϕ. σ_i is a trace state which is not the last trace state. Hence, there are two possible cases:

- if the set of reachable states of $\mathcal{A}_\phi$ is the empty set, then the property will be violated;
- When the last state of σ is reached, there is a loop over this one. $\mathcal{A}_\phi$ is then considered as a graph. Transitions with the formula evaluated to false are deleted. The mapping direct acyclic graph is computed and is browsed in order to determine if an accepting state is reachable infinitely often. Two sub-cases are possible:
  - An accepting state of $\mathcal{A}_\phi$ is reachable infinitely often. Hence the property is satisfied until the end of trace. However, it could be violated after this point of the programme execution. This is the case of safety and liveness properties.
  - Any accepting state of $\mathcal{A}_\phi$ is infinitely often reachable. The property is not satisfied. However, it would be satisfied after this point of the programme execution. This is the case of liveness property.

4.3 Statistical Büchi automaton, a pragmatic approach

We propose a pragmatic approach to handle "end of trace". We provide the user with statistical information on the satisfaction of the property. Two kinds of information are provided:

- Additional information for all properties which have not been verified. When the property is not verified, the last state where the property is true is provided. This information helps targeting where the potential problem is inside the code.
- Additional information for properties which comply with a pattern. This second aspect allows the giving of a set of information which is richer than the other one, because information is given even if the property is evaluated to true.

Statistical information depends on the pattern and is computed using an automaton which is executed instead of the Büchi automaton.

The definition of the statistical Büchi automaton is split into two parts hereafter. The first part consists of the statistical counters. The second part consists of their integration into a classic Büchi automaton.

Definition 7. $\mathcal{C}$ is a set of integer variables called counters. $\Lambda_\mathcal{C} : (\mathcal{C} \to \mathbb{Z}) \to (\mathcal{C} \to \mathbb{Z})$ is a set of actions on $\mathcal{C}$, depending on current value of all variables. Operations are characterized by the following grammar:

$\langle$ operations $\rangle$ ::=
| $\langle$ operations $\rangle$ $\langle$ operation $\rangle$

This rule defines a list of operations. The list can be empty or not.

⟨ operation ⟩ ::= ⟨ counter ⟩ = ⟨ expression ⟩

This rule defines an operation. An operation always modifies a counter.

⟨ expression ⟩ ::= ⟨ ZConst ⟩
| ⟨ counter ⟩
| ⟨ expression ⟩ + ⟨ expression ⟩
| - ⟨ expression ⟩
| ⟨ expression ⟩ × ⟨ expression ⟩
| ⟨ expression ⟩ ÷ ⟨ expression ⟩
| ⟨ expression ⟩ % ⟨ expression ⟩
| Min(⟨ expressions ⟩)
| Max(⟨ expressions ⟩)

This rule defines an expression. Integer constants (*ZConst*) and *counter* can be used inside operations. Authorized operations are addition, opposite, *plication, euclidean division, modulo, minimum and maximum.

⟨ expressions ⟩ ::= ⟨ expression ⟩,⟨ expression ⟩
| ⟨ expression ⟩,⟨ expressions ⟩

This rule is used for Min and Max operations. The list has at least two expressions.

Hence, a statistical Büchi automaton is :

Definition 8. A statistical automaton is a five-uplet $\mathcal{A} = (Q, \Sigma, \rightarrow, q_0, F, \mathcal{C}_0)$ where:

- Q is a set of states
- Σ an alphabet
- $\rightarrow \subseteq Q \times \Sigma \times \Lambda_{\mathcal{C}} \times Q$ a transition relation
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ a set of accepting states;
- $\mathcal{C}_0 : \mathcal{C} \rightarrow \mathbb{Z}^{\iota}$, is a function which returns the initial value of each element of $\mathcal{C}$

Statistical information depends on the pattern and is computed using an automaton which is executed in parallel with the Büchi automaton.

Example 9. *The automaton of the pattern $\Box\Diamond p$ allows the computing of the number of states where p is true (Figure 4). Each time p is true, the nb_p counter is incremented. At the end of trace execution, we know how many times p was true.*

In this pragmatic approach, the statistical automaton replaces the classic Büchi automaton. Each time a transition is true, the mapping list of operations is applied to the counters. Currently, three patterns are recognized (these patterns have often been encountered in the studied industrial applications). The past version of each pattern is recognized to

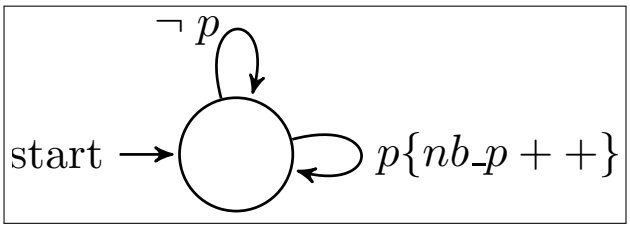

Figure 4: Automaton of the pattern $\Box\Diamond p$

- $\Box\Diamond p$
- $\Box(p \Rightarrow q)$
- $\Box(p \Rightarrow \Diamond q)$

Adding a new pattern with this approach is facilitated. Actually, each pattern is described inside a file (formula, statistical automaton, printing system of statistical information). Hence, if an agent wants to collect additional information for a new pattern, he can define the automaton attached to that pattern to compute his own information.

The next step of this work consists in testing our approach on real industrial cases.

5 Implementation and Experiments

In this section, we detail some elements of the implementation of the prototype *AnTarES*, and propose experiments on real industrial cases. Let us recall, *AnTarES* is an industrial proprietary prototype.

5.1 *AnTarES*, a tools for Analyse of Trace of Execution of Software

AnTarES implements our approach from the transformation of the temporal property into a Büchi automaton to the verification of this property on an execution trace. *AnTarES* uses *Ltl2ba* for the transformation. The major steps are summarized in figure 1

AnTarES is written in OCaml with around 16,300 lines of code. The verifier is split into three applications:

- a reader module, which reads the trace. This task can be done by several readers dispatched on several networked computers. This fact allows the distribution of the calculus load on available resources. In addition, two trace formats are currently handled by the tool: the standard VCD format (Value Change Dump - ASCII based format for dump files), and a data base format. Because these trace formats are transformed into a generic format which is used for verification, adding a new format is easy. The stored trace σ is a sequence of states σ_j. A state is a list of pairs (variable, value);
- a central server module, which will be able to ask the appropriate reader module, if the reading task is dispatched on several computers;
- a client module, which does the verification.

In order to evaluate our prototype, experiments have been done on traces which come from different avionics software. The goal of these experiments was to check that identified requirements can be formalized using our specification language and to assess the efficiency of our approach, for all kinds of properties. As a reminder, verification of temporal properties with non-formal static analysis (code reviewing) needs several days. We aim to decrease the verification time.

Our approach has been tested on programs in order to evaluate the different aspects of this one. The first program is an embedded software, whereas the last one is a hardware model of the dynamic-analysis framework we use to generate execution trace for the first program. The experiments conditions are identical for all experiments : the used machine is an octo-core machine with a 2Ghz-and-512ko-cache processor.

5.2 Experiments on a first software

This experiment evaluates three aspects of our approach: reverse reading, parametric properties, and computation of additional information for a specific pattern. This software is based on the Arinc 653 standard. The verified part of this software is around 550 lines of C code. The Arinc 653 library is around 2,600 lines of C code. It is a multi-task software which is stopped after a finite number of loops. The software communicates with other applications using dynamically created ports. The property consists in ensuring that each port has been initialised before being used. The size of the trace is relatively small (about 129 states). The property is verified for all possible values of *used_port* which are 1,2, and 5 in the trace. The formalized properties are the shape of $\boxdot(used\_port = x \Rightarrow \diamondsuit(created\_port = x))$, where $x \in \{1, 2, 5\}$. The recognized pattern is $\boxdot(P \Rightarrow \diamondsuit Q)$. Then, the mapping statistical automaton computes the number of occurrences of P ($used\_port = 1$ for example) before Q occurs ($created\_port = 1$).

Table 1: Additional information: Occurrence of used port

Port	Occurrence
1	107
2	2
5	20

Table 2: Computation time

Module	Time (s)
Reader	0.015
Server	0.024
Client	0.07

Execution times of each module are displayed in Table 2. Because of the small size of the trace, only one reader module is used here. This is in charge of the trace reading, the server makes the link between the reader and the client, and the client is responsible for the verification itself. Verification time of the client includes transformation of the LTL formula into a Büchi automaton, duplication of the Büchi automaton and verification of the formula. Verification of this property on a trace with 129 states is efficient, considering that four properties are simultaneously verified: one for each value of *used_port*.

According to table 1, which gather the results of the verification of the properties on the trace, we can see the usage of each port. Relative to the number of elements, port 1 is used for $82,9\%$ of the cases, whereas port 2 is used for $1,6\%$ of the cases.

This experiment shows that statistical information can be used to check orthogonal properties, such as the number of uses of a port. Indeed, with this statistical information, the used port and their frequency of use are known.

5.3 Experiments on a second software

This experiment has been done on a model of the AIRBUS simulation framework. This model consists of a processor, an interruption control system, an EEPROM, an emitter, and a watchdog. This model represents 675 lines of C. This is a mono-task software which loops infinitely. The aim of this study consists in verifying the behaviour of the watchdog. It checks time between two writing actions on the EEPROM. If the difference between the two writing actions is more than T_0, then an interruption signal is sent to the calculator and the emitter.

In this experiment, we want to verify that a variable must be written in the EPPROM at lest every T_0 seconds. If time is more than T_0, a watchdog raises a signal to the controller, otherwise

nothing happens.

The property can be split into two sub-properties which are:

- $\square((\tau - last\_write\_time > T_0) \Rightarrow signal)$, means when the delta between the last write time and the current time (τ) is upper than T_0, then *signal* will be raised (overflow case).
- $\square((\tau - last\_write\_time \leqslant T_0) \Rightarrow \neg(signal))$ means when the delta is lower or equal to T_0, then *signal* will not be raised (nominal case) .

The properties have been verified on a trace with $1,848,633$ states. In addition, the equations follow the pattern $\square(p \Rightarrow q)$. Hence, we have information about how much the overflow case and the nominal case happen. We cut trace reading in 5 reader modules.

Table 3: Second software, time results

Module	Nominal Time(s)	overflow Time(s)
Reader(min-max)	9.21-10.09	9.21-10.09
Server	3.04	3.04
Client	25.84	35.84

Pattern information says that the nominal case happens $1,848,629$ times (more than $99,99\%$ of the cases) and overflow case happens only 4 times ($2.16 \times 10^{-4}\%$). This information is essential, because it shows that overflow is not frequent. If overflow appears more frequently, then the software will probably have a bug. The acceptable maximal level of appearance of this second property depends on the size of the trace and on the software. This kind of problem could not be detected without statistical information.

Finally, this experiment shows that verification of temporal properties on big traces including computation of statistical information is efficient (less than 50 seconds, trace reading included). In addition, use of patterns can help to detect bugs which are complex to find.

5.4 Discussions around the experiments

The experiments have been processed on embedded avionics software and on models of the dynamic-analysis framework. It shows that the defined approach is efficient on big traces. Indeed, the order of magnitude of time verification is several hundred seconds in comparison with several days with a non-formal verification method (code reviewing). Use of patterns to compute statistical information is useful to help decide if a property is true or false when the entire trace has been analysed. In addition, this statistical information allows the treatment of orthogonal properties such as "how much time a port has been used?" (first software).

Conclusions

This paper has presented an approach for the verification of temporal properties on execution traces of avionics software. This approach was needed because these kinds of properties are difficult to verify using existing techniques. The definition of the solution was guided by industrial needs and constraints. This approach proceeds in three steps: definition of a dedicated property language, generation of trace executions using static analysis, and verification of the property on the trace. The verification step is built in order to handle finite traces, whereas temporal properties have a semantics on infinite traces. To do this, additional statistical information about the trace are provided to help the verifier to conclude about the verification result. This statistical information depends on the property pattern. Frequently encountered patterns are implemented in *AnTarES*. Other patterns can easily be added with the definition of a new statistical automaton.

A prototype *AnTarES* has been developed (18,500 lines of OCaml code) and integrated into the AIRBUS simulation framework. Experiments conducted on industrial applications allowed us to assess the efficiency of our approach on different kinds of properties and different sizes of trace. An industrial deployment of the tool would require adding a graphical user interface to display results and to provide user-friendly ways to write properties. In addition, in order to simply the specification of properties we aim to improve the language with parametric variable handling. We hope to improve the handling of past formulae by reverse reading the trace during the automaton execution. These are parts of future work.

This approach is a contribution to the overall industrial strategy, which consists in reducing the part of testing in the verification process of avionics software by developing and integrating static and dynamic analyses techniques [18]. Future work will target means to better combine static and dynamic analysis.

Acknowledgment

The work is funded by the French national research agency (ANR) in the context of the PERFECT and RE(H)STRAIN Projects.

Bibliography

[1] Runtime verification, 2001-2009.

[2] H. Barringer, A. Goldberg, K. Havelund, and K. Sen (2004); Rule-based runtime verification. In B. Steffen and G. Levi, eds., *Verification, Model Checking, and Abstract Interpretation*, vol. 2937 of *LNCS*. Springer.

[3] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen (2003); Eagle does space efficient ltl monitoring. Technical report, Nasa.

[4] Andreas Bauer, Martin Leucker, and Christian Schallhart (2011); Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64.

[5] J.Richard Büchi (1990); On a decision method in restricted second order arithmetic. In Saunders Mac Lane and Dirk Siefkes, editors, *The Collected Works of J. Richard Büchi*, Springer New York., 425–435.

[6] Marcelo d'Amorim and Grigore Rosu (2005); Efficient monitoring of omega-languages. In *CAV'05*, 364–378.

[7] Doron Drusinsky. The temporal rover and the atg rover. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*. Springer, 2000.

[8] A. Ferlin and V. Wiels (2012); Combination of static and dynamic analyses for the certification of avionics software. In *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, 331–336.

[9] Antoine Ferlin (2013); *Verification de propriétés temporelles sur des logiciels avioniques par analyse dynamique formelle*. PhD thesis, Thèe de doctorat dirigée par Wiels, Virginie Sureté de logiciel et calcul de haute performance Toulouse, ISAE 2013.

[10] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, vol. 2102 of *LNCS*. Springer, 2001.

[11] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering*, 2001.

[12] K. Havelund and G. Rosu (2001), Monitoring programs using rewriting. In *Automated Software Engineering*, 135 – 143.

[13] Klaus Havelund, Grigore Rosu (2002), A rewriting-based approach to trace analysis. *Automated Software Engineering*, 1-21.

[14] S C Kleene (1956), Representation of events in nerve nets and finite automata. In *In Automata Studies*. Princeton University Press: Princeton.

[15] Martin Leucker, Christian Schallhart (2009), A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS07).

[16] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, Grigore Rosu (2012), An overview of the mop runtime verification framework. *International Journal on Software Tools for Technology Transfer*, 14:249–289.

[17] A. Pnueli, A. Zaks (2006), Psl model checking and run-time verification via testers. In *FM 2006: Formal Methods*, vol. 4085 of *LNCS*. Springer.

[18] Jean Souyris, Virginie Wiels, David Delmas, Hervé Delseny (2009), Formal verification of avionics software products. In *Formal Methods, Lecture Notes in Computer Science*, 5850: 32-546.

[19] Volker Stolz and Eric Bodden (2006), Temporal assertions using aspectJ, Proceedings of the Fifth Workshop on Runtime Verification (RV 2005), *Electronic Notes in Theoretical Computer Science*, 144(4):109 –124.