Formal verification of SAM state machine implementation

A. Stéphane Duprat[1], B. Pierre Gaufillet[2], C. Victoria Moya Lamiel[1], D. Frédéric Passarello[1]

1: Atos Origin, 6 Impasse Alice Guy, B.P. 43045, 31024 Toulouse Cedex 03
2: Airbus Operations S.A.S., 316 route de Bayonne, 31060 Toulouse Cedex 9

Abstract: This paper reports the results of an experiment about the formal verification of source code made according to an EMF model. Models define the semantics of a system whereas the source code defines its implementation. We applied this solution to a model of Automaton in SAM language and its C language implementation. The technical environment is close to an industrial operational context and all the tools are available. The experimentation has succeeded and has to be consolidated with bigger cases before an introduction in the operational development process. More generally, this solution must be extended to other model languages.

Keywords: proof, formal verification, SAM, Caveat, Frama-C

1. Introduction

In the context of critical software development, formal verification is a way of improvement of verification activities that is cost effective and with a high degree of quality. This kind of functional verification of programs requires a formal specification to be defined by the user.
Another way of improvement in the software development process is the use of models in specification and design. Among other things, one advantage of model approach is that it helps the user to define its requirements in a more comprehensive and formal way.
We will try to take advantage of both techniques that are formal verification and model driven engineering by choosing a most formalized kind of model and considering it as the formal specification in a proof verification of program process.
The context of this study is close to the operational context: the chosen model language is SAM, that is already used, source code is compliant with critical software development constraints and formal verification tools of C language that are Caveat [1] already used for unit proof [3] and Frama-C/Jessie [8].
This study will first demonstrate feasibility of this solution. Then, we will focus on performance of different solutions defined by the implementation choices of the source code and the proof verification. We will finally make a balance of this method, comparing it with case test generation and source code generation.

2. Industrial context

The development of avionics application is generally considered as following the classical V cycle: specification of the functional needs, design of the solution, implementation of the product, and careful verification of the product against its design and specification. Of course, the validity and the consistency of each sub-product are also verified. This approach, driven by the DO-178B document, often implies a huge effort of verification.

A quite common situation is to develop 1000 lines of automated tests for only 100 lines of code. To improve this situation, several improvements are possible:

- Replace the implementation and associated tests by a qualified code generator: very effective once the generator has been developed, it is also very expensive, as the code generator needs to be developed with the same DO-178B constraints than the target application.
- Develop a test plan generator: this solution is a light one compared to the development of a code generator, as it has to be qualified as a verification tool – verification tools have only to be verified against their functional specifications – rather than a development tool. But test plans are de facto limited. They can only cover some well known scenarii. Test plan generators often – but not necessarily – rely on a test engine.
- Develop a formal properties generator: this solution remains a light one. This generator has only to be qualified as a verification tool. But the formal verification engine on which it is based is comprehensive: each property is verified against every possible execution of the code.

Of course, the 2 last approaches require a test or a verification engine compatible with DO-178B requirements. But fortunately, it is already the case (note that these tools also need to be qualified, or their results have to be reviewed).

Code generation is now used for years, but because of their cost, it is not adapted to every context. Test plans generation is on its side also already used successfully in industrial contexts, but the more innovative formal properties generation remains more seductive because of its completeness. This study therefore focuses on it. As our goal is to prepare a deployment in the short term, our use case has been extracted from a real development involving networking protocols engineering. The specification of this networking protocol has been formalized as simples Mealy machines using SAM language ([10]) – SAM is often used to help with functional split up of applications, but it also allow to describe formal behaviours with automata. This networking protocol has been implemented using C, in compliance with common constraints of the embedded domain (no malloc/free, etc.).

3. Principles

Methodological Objectives

The main methodological objective is to formally verify that an implementation conforms to its specification formally defined in a model.
The expected benefit is an easier and better validation of the source code made with a model driven approach.
We are at the beginning of this approach and we have limited the scope of this study to finite state machine which is one of the most formalized behavioural model. The exact format of the model handled is defined by the SAM "Automaton".
The model of the state machine is considered as the specifications to validate. The model language should have a well defined semantic upon which both the code and the verification will be built. For example, in a classic approach by unit testing, scenarios are generated or manually defined from the model.
The proposed solution consists in replacing tests by formal verification. That means that we will replace the source code of unit tests by formal properties. These properties will be written in the format of a formal verification tool like Caveat or Frama-C.

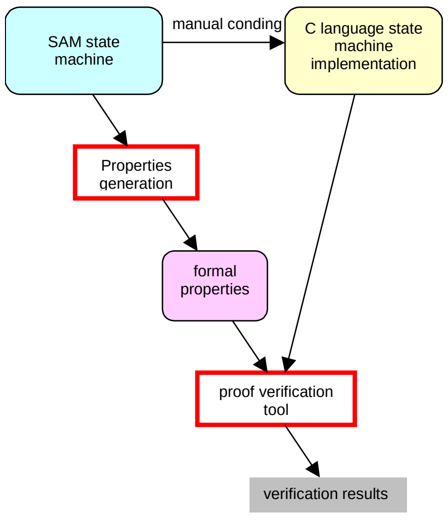

Figure 1 : Principles of the formal verification of C source code

To validate this strategy, we have to succeed in the two following challenges (see figure #1):
1/ Describe the behaviour of the SAM state-machine as a set of formal properties;
2/ Make the proof of the formal properties for a given implementation of the state-machine.

The first objective requires both a methodological work to define the format of the formal properties and to develop the tools which will generate automatically these properties.
The second objective requires the use of a verification tool and its success depends on both generated properties and source code.

Definition of formal properties

The logic defined by an automaton is implemented in our context by a single function (called the "**transition function**") which is in charge of computing the output values and the new state according to the input values.
Input and output values are booleans which are usually coded as integers (they can be set with either "0" – false - or "1" – true).
The transition function is called in a loop where a translator sets the inputs from the external environment of the Automaton. An actuator reacts to outputs to execute specific actions (callbacks).

This architecture is described in the figure #2 below:

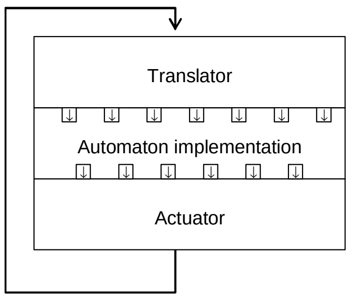

Figure 2 : Architecture of a SAM automaton

A main advantage of this architecture is that the semantic of the Automaton is strictly implemented by one function with booleans operands.
This function has operands mapping with inputs and outputs of the state machine and one operand implementing the current state.
Based on this architecture, the chosen solution consists in a unitary verification by proof of the implementation of this unique function.
The unitary proof principles are defined in [3]. In our case study we use the unitary proof to check the relation between inputs, outputs and the current state of the automaton.
The mapping between elements of the model and terms in the property language is defined as follows:

Name of Automaton → Name of the function
Name of InCtrlPort → Input name
Name of OutCtrlPort → Output name
Name of current_state = state

The unitary proof method is an exhaustive verification method.
In this context, that means that whole properties will define the expected value of each output port and of the new state from the values of the input ports and of the current state.
There are no theoretical difficulties in these definitions as SAM Automatons have a well-defined determinist semantic.
The chosen solution consists in:
PRECONDITION

- Define range of inputs
  (ex : "IN1 ∈ {ON,OFF};")
- Define range of current state
  (ex: "state ∈ {S0, S1, S2, S3};

POSTCONDITION
For all states with a transition, define:

- The new value of the state
- Values of outputs activated by the transition
- Values of unmodified outputs

Example (Caveat syntax):

```
Post P1: state'=S1 ∧ (I1=ON ∨ I3=ON) => state=S2;
Post P2: state'=S1 ∧ (I1=ON ∨ I3=ON) => O2=ON;
Post P3: state'=S1 ∧ (I1=ON ∨ I3=ON) => O1=O1';
```

This solution produces an exhaustive definition of outputs and could be read by the user.
The semantic definitions is splitted into several different properties, and in case of one or more bugs in the source code, the unproven property(ies) will allow to identify accurately the origin of the bug(s).

4. Implementations

4.1 Formal Verification tools

Caveat is a formal verification tool for C ANSI programs. This tool can be used to perform functional verifications by proof of program. Caveat is based on the Hoare logic as for the theoretical background.
The user first defines formal properties in a first order predicate language. Post-conditions are a kind of properties defining relations between inputs and outputs operands of a function. These properties are applied at the end of a function. As opposite to the Postconditions, the Preconditions are applied at the beginning of a function. They are considered as hypothesis for this function and in the case where this function has at least one caller, the Precondition must be verified in the caller context.

Caveat has its own demonstrator that is called when the tool is asked to prove a property. After computation, the property is considered as verified if the result of the demonstrator is true. If it is not the case, the tool produces the *remaining* of the property: the remaining is the condition required to prove the property. A failed proof can be caused by either a real inconsistency between the source code and the property or just a weakness of the tool.
In order to determine if the property is really false, the user can terminate the proof of the property with an interactive module.
Caveat has been recently improved by the adjunction of a gateway to the multi-prover platform why ([9]). This feature enable Caveat to execute an external prover such as alt-ergo [11], simplify [12], z3 [13] to prove its proof obligations. This multi-prover gateway is well implemented at a low level and enables to combine both use of the interactive features of Caveat and the power of external provers.
The user is free to use these verification tools with its own methodology. But some important point is that one of this tool (Caveat) has been integrated in the process development of critical software in place of unit test. This usage is called unit proof and is described in a previous article [3].

Frama-C is a framework for static analysis of C programs. It is an extensible framework where plugins can work in a collaborative way. The combination of different techniques of analysis implemented through different plugins produces a powerful and multi-purpose tool.
The main components of Frama-C are currently the Value Analysis and Jessie. Only the last one has been used for this study.
Jessie [8] is a Frama-C plugin used for formal verification like Caveat. This plugin is based on the **why** platform.
Properties are introduced as annotations in the comments of the source-code. The language for annotations is called ACSL [6] and is also the language of the Frama-C platform. ACSL is a very rich language to specify semantics of the program, but not all the tools are able to handle all the features of this language.
As Jessie is based on the why platform, it gets advantages of the multi-provers platform.

Frama-C doesn't have the same limitations than Caveat as for the semantic of the pointers and should be able to tackle a larger variety of programs than Caveat. Caveat is older than Frama-C but it has a higher maturity and more limitations. Caveat is already deployed in industrial projects.

4.2 Modus Operandi

We want to check that a given implementation of a SAM automaton conforms to its specifications described in a SAM model.

We can manually define the verification properties which will be checked by static analysis tools like FRAMA-C and Caveat. These properties are set either as annotations in the automaton source code (FRAMA-C/Jessie) or into a separate file (Caveat).

However, writing verification properties manually is tedious and error-prone, for even the smallest and simplest of SAM automatons. Therefore, we can develop a tool – a property generator - to handle this task.

The SAM model is used to generate the formal properties. However, the format of these properties also depends on the implementation choices (functions declarations/interface, variable names, types…).

The SAM model is an EMF model which has been edited under TOPCASED. It can be visited by using the EMF API to retrieve the automaton's elements in order to generate the properties.

The algorithm used to generate the verification properties is made of two main steps:

- Step 1: the generator first visits the SAM model to:
    - o Gather all needed information about the automaton's states and transitions ;
    - o Convert the triggering conditions in ACSL/Caveat languages;
- Step 2: the tool generates the verification properties according to both information found in the first step and coding conventions.

To describe this algorithm we will use an example of a SAM automaton (see figure #3) which is assumed to be contained into a SAM model.

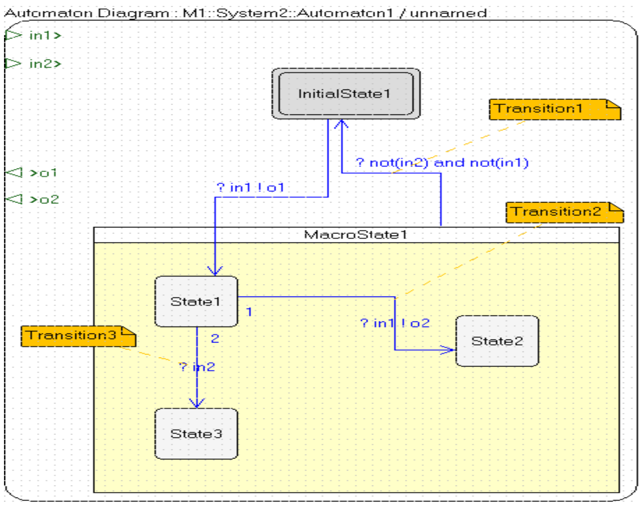

Figure 3 : a trivial SAM automaton

Moreover, the implementation of the SAM automaton is presupposed to use global variables in order to hold the current state, input and output ports of the automaton.

```
 typedef enum { // state enumeration
  STATE_InitialState1,
  STATE_State1,
  STATE_State2,
  STATE_State3
}XXX_Te_StateList;
extern XXX_Te_StateList state; // current state
extern int IN1; // input port #1
extern int IN2; // input port #2
extern int O1; // output port #1
extern int O2; // output port #2
```

The transition function looks like this:

```
void XXX_Se_Automaton1(){
  switch (state){
  case STATE_InitialState1 :
    {
      if (IN1){
        state=STATE_State1;
        O1=1;
      }
      break;
    }
  case  STATE_State1 :
    {
      if (!IN1 && !IN2){
        state=STATE_InitialState1;
      }else if (IN1){
        state=STATE_State2;
        O2=1;
      }
      else if (IN2){
        state=STATE_State3;
      break;
    }
    }
  case  STATE_State2 :
    {
      if (!IN1 && !IN2){
        state=STATE_InitialState1;
      }
      break;
    }
  case  STATE_State3 :
    {
      if (!IN1 && !IN2){
        state=STATE_InitialState1;
      }
      break;
    }
  }
}
```

We will concentrate on the state named "State1" in order to study the management of the SAM transitions priorities.

Formal properties generation algorithm: Step 1

For each Automaton found in the model:

1. For each state existing in the automaton, retrieve all the outgoing transitions:

    a. Sort the outgoing transitions according to their priority. The priority of a SAM transition depends on both its priority index and the existence of one or more macro states (macro state outgoing transitions have a higher priority);

```
For state "State1" (highest to lowest):
- **Transition1** (macro-state outgoing transitions
have a higher priority)
```

```
- Transition2 (lower priority index = highest
priority)
- Transition3
```

b. For each transition:

i. Translate both the conditions which trigger the transition (state of input ports) and the emission (state of output ports) from the SAM syntax to either ACSL or Caveat syntax (using ANTLR). We must indicate to the prover that we consider the value of the input port prior to the call to the transition function. This is done by using the function “\old()” in Jessie and a quote in CAVEAT:

```
Transition1: not(in2) and not(in1)
 - ACSL: !(\old(IN2)==1) && !(\old(IN1)==1)
 - Caveat: ¬(IN2'=1) ∧ ¬(IN1'=1)
Transition2: in1
 - ACSL: \old(IN1)==1
 - Caveat: IN1'=1
Transition3: in2
 - ACSL: \old(IN2)==1
 - Caveat: IN2'=1
```

ii. Append the negation of the triggering conditions of higher priority transitions to the triggering conditions of the current transition. This is to take into account the priorities. These conditions have been added to the properties and are indicated, in bold:

```
Transition1:
 - ACSL: !(\old(IN2)==1) && !(\old(IN1)==1) (no
transition with higher priority)
Transition2:
 - ACSL: \old(IN1)==1 && !(!(\old(IN2)==1) &&
!(\old(IN1)==1))
Transition3:
 - ACSL: \old(IN2) && !(!(\old(IN2)==1) &&
!(\old(IN1)==1)) && !(\old(IN1)==1)
Transition1:
 – Caveat: ¬(IN2'=1) ∧ ¬(IN1'=1) (no transition
with higher priority)
Transition2:
 – Caveat: (IN1'=1) ∧ ¬(¬(IN2'=1) ∧ ¬(IN1'=1))
Transition3:
 – Caveat: (IN2'=1)  ∧ ¬(¬(IN2'=1) ∧ ¬(IN1'=1))
∧ ¬(IN1'=1)
```

iii. Store the output ports which are **not** changed during the transition:

```
Transition1: O2
Transition2: O1
Transition3: N/A
```

c. Store forbidden transitions (transitions which must never happen for the current given state).

```
 - None for the state « State1 »
 - From State2 to State1
 - From State2 to State3
 - From State3 to InitialState1
 - From State3 to State2
```

Formal properties generation algorithm: Step 2

The generation of the verification properties follows the algorithm described below.

For each state:

1. Generate the verification properties which check all outgoing transitions as shown in the pseudo code below:

```
old(state)==<starting_state)>
&& <triggering conditions for this transition>
&& !<∑(triggering conditions for transitions of
higher priority)>
    ➔ state=<destination_state>
    && <emitted output ports of this transition
are true>
    && <non-emitted output ports remained
unchanged>
```

For instance, the verification property for the Transition #2 of our example (ACSL code):

```
\old(state)==STATE_State1
&& (IN1==1) && !(!(IN2==1) && !(IN1==1))
➔ state==STATE_InitialState1
&& OUT2==1
&& \old(OUT1)== OUT1;
```

2. Generate the properties for the combinations of inputs which do not trigger any transition (and thus neither change the current state nor the output ports). Here we use a little trick: it is more simpler to retrieve the negation of all the combinations of inputs which effectively trigger a transition:

```
old(state)==<starting_state>
&& !(∑<triggering conditions for all possible
transitions))>
➔ state=<starting_state>
&& <all output ports remain unchanged>
```

3. Generate the properties which check than non existing “transitions” never happen (this is redundant with the previous verification but it may help the user in finding bugs):

```
old(state)=<starting_state>
➔ state!=<forbidden_state>
```

4.3 Presentation of the property generator

The properties generator tool implements the algorithm which has been presented in the previous paragraphs in order to generate the verification properties from a given SAM model.

As the model is an EMF model, the properties generator has been coded in Java within the Eclipse platform.

The trickiest part is certainly to write the properties according to the targeted source code. Indeed, the formal properties must refer to the interface elements of the source code (global variables, structure fields, etc...) which should be compliant with the coding conventions of the project. The format of the formal properties thus tightly depends on the implementation of the SAM automaton.

Many kinds of C implementations of a SAM automaton may exist and this thus makes the generation process more difficult:

- In the simplest case, the state of the SAM automaton and the states of both input and output ports may all be implemented as **global variables** in the source code. These global variables are read and written by the transition function;
- The current state of the SAM automaton, of its inputs and its outputs may be maintained in a **C structure** which is given as a non-constant parameter of the transition function;
- The input and output ports may be coded either **as integers or as bit flags**;
- Finally, the transition function itself may call other functions to read or write the input and output ports, to change the automaton state, etc… Under FRAMA-C/Jessie, these functions must be annotated as well.

The impact of the automaton implementation is in fact broader and may concern the verification process as well, as most provers do not support pointers to functions (for instance) or may not implement all necessary semantics (bit flags for instance).

Further enhancements have been made to the tool to make the properties more human readable:

- It is possible to generate a single property for each targeted state and for each emitted or non-emitted output port.
- The transition's conditions can be factorized in one way or another, depending on the prover:
    - o In ACSL (FRAMA-C/Jessie), it is possible to group properties into *behaviours*. Such behaviour is used to define a set of conditions which hold true for all subsequent properties. These conditions are described in an "assume" property under the name of the behaviour: in our case, the "assume" property is used to mutualise the starting state:

```
 behavior state_INIT:
   // The starting state of the “state_INIT”
   // behaviour is STATE_INIT
 assumes Automaton->CurrentState==STATE_INIT;
   // The following properties assume that
   // the starting state is STATE_INIT
 ensures from_INIT_to_STATE1_1:
inSet(\old(*Automaton), INPUT_PORT1) ==>
   Automaton->CurrentState==STATE1;
 ensures from_INIT_to_STATE1_2:
inSet(\old(*Automaton), INPUT_PORT1) ==>
   outSet(*Automaton), OUTPUT_PORT1);
```

The *inSet* and *outSet* are logic function defined in ACSL helping input/output status access.

- o For Caveat, a definition (in a 'LET' term) holding all the triggering conditions can be defined prior to the properties which will use that variable:

```
LET cond_from_INIT_to_STATE1=Automaton'.[*',
.CurrentState]=TCA_E_INIT ∧
((bit&32(Automaton'.[*', .InputEvents],
TCA_E_RD_FILE_REQ)));
   Post from_INIT_to_STATE1_1:
cond_from_INIT_to_STATE1 => Automaton.[*,
.CurrentState]=STATE1;
   Post from_INIT_to_STATE1_2:
cond_from_INIT_to_STATE1 => bit&32(Automaton.[*,
.OutputEvents], OUTPUT_PORT1);
```

5. Results

5.1 Proving capabilities

We have tested both the verification tools and the property generator on various SAM automatons and on different kinds of C implementations. We will study here two automatons:

- **A trivial case of a SAM automaton.** This is the case described previously (see figure #1);
- **A SAM automaton from an industrial project**. The default implementation was made regarding a previous SAM model without priorities defined between transitions. Consequently, this automaton had a lot of unconformities **that have been detected by Caveat** (and its external provers). The existence of these bugs has been particularly useful to check the reliability of the verification tools. The source code of the SAM automaton has been fixed to further study the behaviour of the verification tools.

The first expected quality of a verification tool is to detect bugs and not to prove a false property.
In an industrial context, the tool should be able to prove all the true properties.

The following table indicates the percentage of properties which have been proved by both FRAMA-C/Jessie and Caveat for the various implementations of SAM automatons:

	Trivial case	Industrial case (initial)	Industrial case (fixed)
Jessie/Alt-ergo	100%	X	X
Jessie/Simplify	100%	X	X
Jessie/z3	100%	X	X
Jessie/All provers	100%	X	X
Caveat	100%	37%	57%
Caveat+alt-ergo	100%	49%	100%

Table 1 : Percentages of properties proved by FRAMA-C/Jessie and Caveat (trivial and industrial cases)

In the case of the trivial automaton:

- We can see that the combination of Jessie and any of the tested provers (alt-ergo, simplify and z3) allows to prove 100% of properties;
- The internal demonstrator of Caveat is able to prove 100% of properties (using an external prover is not necessary for a simple case like this one).

In the case of the industrial SAM automaton:

- Jessie results have not been reported since we have encountered some difficulties relative to the implementation of the SAM automaton;
- The internal demonstrator of Caveat is able to detect all the errors in the initial source code but not to prove all true properties.
- Combining the internal demonstrator of Caveat and an external prover (like alt-ergo) is necessary to find all the true properties (49% in the initial source code of the industrial automaton and 100% in the fixed version).

The Caveat GUI can be used to check the failed properties and to re-write their remaining conditions. In our case, rewriting the remaining condition of a failed property led to the exact definition of the inputs combinations that caused the bug. In case of failed proof of a true property, we have easily succeeded in proving it thanks to basics interactive features of Caveat. It is however quicker and simpler to use an external prover like alt-ergo to complete the results of the Caveat demonstrator as this allowed to prove 100% of properties.

As indicated earlier, we encountered some difficulties with Jessie applied to the source code of the industrial SAM automaton. This case study is still under analysing in order to find the real origin of these difficulties (ACSL formal properties, C language structures). We have thus rewritten this automaton with integers and global variables and we succeeded in applying Jessie to this new implementation as shown in the table below:

	Rewritten Industrial case (initial buggy code)	Rewritten Industrial case (fixed)
Jessie/Alt-ergo	61,7%	100%
Jessie/Simplify	61,7%	100%
Jessie/z3	61,7%	100%
Jessie/All provers	61,7%	100%

Table 2 : Percentages of properties proved by Frama-C/Jessie with a rewritten implementation of the industrial SAM automaton

We can observe the following results:

- All the provers were able to detect the failed properties (37/60) in the buggy source code. We have thoroughly checked the wrong properties to ensure that they really incriminate the bugs of the source code. To do so, we were forced to develop a new tool to bind the failed obligation proofs to their corresponding user properties (such information is not provided by Frama-C/Jessie by default);
- All the provers were able to prove all properties in the fixed version of the automaton.

5.2 Performances

Typically, the provers should be run periodically on a dedicated machine: the performance of the code analysis is not the most important criteria when having to chose and deploy a verification tool.
But the quickest is the analysis, the quickest it is possible to fix possible bugs and to re-iterate. And as we are going to see, real performance differences exist between the tools which have been tested:

	Trivial Case	Industrial Case (initial)	Industrial Case (fixed)
Analysis duration (alt-ergo)	< 1mn	31mn	101mn
Analysis duration (simplify)	< 1mn	4mn	57mn
Analysis duration (z3)	6mn	14mn	69mn
Analysis duration (Caveat)	10s	1mn24	1mn31
Analysis duration (Caveat + Alt-ergo)	10s	1mn25	1mn31

Table 3 : duration of the code analysis according to the verification tool and the studied SAM automaton

Caveat offers the quickest analyzes: the verification process lasts only a few minutes, whatever the complexity of the SAM automaton and even when the external provers are enabled. In comparison, FRAMA-C/Jessie analyses are really slow and may easily last more than one hour for the industrial automaton.

This is partly due to the fact that Caveat calls its external provers only when its internal demonstrator has failed to prove a property. Moreover the external provers work with the remaining of the failed property.
But this huge performance gap is mostly due to the inner working of Jessie which generates a lot of proof obligations for each property as we can see in the following table:

	Trivial Case	Industrial Case (initial)	Industrial Case (fixed)
# User Properties - FRAMA-C Jessie	**18**	**66**	**66**
# Proof obligations - FRAMA-C Jessie	807	2919	19567
# User Properties - Caveat	**20**	**60**	**60**
# Proof obligations - Caveat	N/A	N/A	N/A

Table 4 : number of user properties and of the generated proof obligations for FRAMA-C/Jessie and Caveat

We can first observe that the number of user properties is higher for Jessie than it is for Caveat since all functions called by the transition function must be annotated as well in the case of Jessie.
Caveat does not generate proof obligations for each user property whereas for Jessie the number of proof obligations depends on both the user property count and the complexity of the code (number of code branches and number of instructions, both higher in the fixed version of the industrial case). The number of proof obligations generated by Jessie can reach nearly 20000 for the fixed industrial case: this increases dramatically the work of provers and explains why Jessie is slower than Caveat.

Finally, for information purposes, we can note that the FRAMA-C/Jessie GUI is even slower than the command line (both at launch and during the proving process).

6. Conclusion

In this article, we have studied the formal verification of SAM automatons implementations. The formal verification ensures that a given implementation of a SAM automaton conforms to its specifications (described in an EMF model in our case).

Formal verification first consists in defining the properties which describe the behaviour of the SAM automaton. These formal properties are then possibly proved during the analysis of the automaton source code: a failed property may indicate a bug in the implementation.

Two tools allowed us to perform the static analysis: FRAMA-C/Jessie and Caveat. They may use external provers to prove the verification properties (like alt-ergo, simplify and z3).

Both tools have proved all the properties defined for a trivial automaton.
More importantly, Caveat (seconded by the external prover “alt-ergo”) has detected many bugs in a development version of an industrial SAM automaton. After having fixed the code, Caveat finally proved automatically all the properties.
This experience confirms the ability of this solution to detect any unconformity.
The necessary condition returned by Caveat in case of a failed proof is enough comprehensive for the user. It shows concretely the exact combination of entries that causes the failure of the proof.
With this information, the developer can quickly fix the bug and reiterate the formal verification. This process is fully compliant with an operational use.

Frama-C was able to provide the same quality of verification but not with the initial implementation of the automaton.

Caveat offers the best performances but provides a limited support for pointers. The free and open source solution FRAMA-C/Jessie seems promising but some enhancements are needed to make this solution more suitable for industrial projects (better traceability between user properties and proof obligations).

This study has shown that the functional verification of SAM automatons through code analysis may advantageously replace unit tests. Hereafter we enumerate the key points of this strategy.

A direct verification between model and implementation
In this new strategy, we consider the SAM Automaton as the formal specification. The generated properties are temporary elements in the process. In the context of this study, all verifications are proven automatically. Indeed, test efforts are replaced by a fully automatic and formal verification, here, very cost effective.
We have only considered a specific model of state machines, SAM Automatons, that react with boolean input, which is simple problem. It would be valuable to extend this strategy to others and more rich formalisms, like a subset of UML StateMachine or Activity.

Quality of verification
This formal verification provides a higher degree of quality than unit testing because this kind of verification is exhaustive:

- All code branches are visited by the provers during the verification process;
- The exact condition of a transition is checked, not an incomplete set of cases satisfying the condition.
- Formal verification checks the combinations of input ports which does not trigger any transition in the SAM automaton

- Formal verification checks that the transitions which do not exist in the model cannot occur according to the source code;

Comparison with Code generation

This strategy can be compared with another solution equally fully automatic consisting in generating the SAM automaton source code from the model.

In the case of critical software with certification constraints, the certification of a code generator is usually a very huge process. The verification tools presented in this study also have to be certified but certifying a verification tool is easier than certifying a code generator.

It is also possible to combine both techniques, using a qualified verification tool to check source code produced by a non-qualified generator tool.

Less constrained environments may also benefit from formal verification as it is fully automatic and as it provides a good quality of verification.

During this study, we have developed different generators targeting two verification tools (Caveat and Frama-C) and different implementation choices in the source code. If we consider that developing or adapting a property generator is quite easy, it may be possible to adapt our solution to different contexts and projects.

Operational application?

The proposed solution has succeeded with a context close to the operational context.

This solution has to be consolidated by other case studies with higher complexity. Generally, there is always a limit over which tools and methods are ineffective. It is necessary to know this limit before any decision in using this strategy for Automation verification.

7. Acknowledgement

Many thanks to J. Souyris at Airbus for introducing formal verification of program with Caveat in an industrial use, D. Delmas at Airbus for all its works around static analysis, Frama-C and Jessie development teams and researchers at CEA and INRIA for their support.

8. References

[1] C. Antoine, A. Trotin, P. Baudin, J.M. Collard, J. Raguideau: "*Caveat: a Formal Proof Tool to Validate Software*", IAEA Conference, IAEA'94, Helsinki, May 1994.

[2] V. Prevosto: "*Specification and Proof of C Programs using ACSL and Frama-C*", CEA LIST, Avignon, France.

[3] Stéphane Duprat, Jean Souyris, Denis Favre-Felix: "*Formal verification workbench for airbus avionics software*". ERTS, Toulouse, 2006.

[4] Patrick Baudin, David Delmas, Stéphane Duprat, Benjamin Monate: "*Proving Temporal Properties at Code Level for Basic Operators of Control / Command Program*". ERTS, (Toulouse, France), 2008.

[5] F. Randimbivololona, J. Souyris, P. Baudin, A. Pacalet, J. Raguideau, D. Schoen: "*Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach*" , World Congress on Formal Methods in the Development of Computing Systems, FM'99, Toulouse, September 1999.

[6] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto: "*ACSL: ANSI/ISO C Specification Language (V1.4)*", http://frama-c.com/download/acsl_1.4.pdf, 2010.

[7] Loïc Correnson, Pascal Cuoq, Armand Puccetti, and Julien Signoles: Frama-C User Manual, November 2009. http://frama-c.cea.fr/download/user-manual-Beryllium-20090902.pdf.

[8] Claude Marché, Yannick Moy: Jessie Plugin Tutorial, February 2010. http://frama-c.com/jessie/jessie-tutorial.pdf.

[9] Jean-Christophe Filliâtre and Claude Marché. "*The Why/Krakatoa/Caduceus platform for deductive program verification*". In Werner Damm and Holger Hermanns, editors, 19th International Conference on Computer Aided Verification, Lecture Notes in Computer Science, Berlin, Germany, July 2007. Springer-Verlag.

[10] D. Thivolle, H. Garavel, X. Clerc. : "*Présentation du langage SAM d'Airbus*", Technical report, INRIA/VASY , 2008.

[11] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. "*CC(X): Efficiently Combining Equality and Solvable Theories without Canonizers*". In 5th International Workshop on Satisfiability Modulo, Berlin, Germany, July 2007.

[12] D. Detlefs; G. Nelson; J.B. Saxe: "*Simplify: A Theorem Prover for Program Checking*", HP Laboratories Palo Alto, July 2003.

[13] L. de Moura and N. Bjørner : "*Z3: An Efficient SMT Solver*". In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary, 2008.

9. Glossary

PDF:	Portable Document Format
ACSL:	ANSI/ISO C Specification Langage
SAM:	Structured Analysis Model