A Realistic Involvement of Formal Methods

D. Bégay[1] / A. Rauzy

CNRS-LaBRI / MVTsi

351, cours de la Libération

F 33405 Talence cedex

Tel. + 33 5 56 84 60 83

fax + 33 5 56 84 66 69

{begay,rauzy}@labri.u-bordeaux.fr

[1] Didier Bégay is now with France Telecom /CNET, “Formal Methods and Logics” Group, 2 avenue Pierre MARZIN, F 22307 LANNION; didier.begay@cnet.francetelecom.fr

Summary: In this article, we report a real-life application of so-called “Formal Methods”. The part of the project we were involved in was to verify that an embedded circuit satisfies a safety property. We describe the circuit as well as the mathematical and computer tools we used. We discuss methodological issues and we present some of the various experiments we performed. Finally, we draw some general conclusions about the practicability of formal verification techniques.

Keywords: embedded circuits, formal verification techniques, model-checkers, proof assistants.

Introduction

For more than a decade, so called “Formal Methods” have raised hopes that it is possible to design techniques to make software development safer, more reliable and less expensive. The existence of adequate mathematics and powerful computing resources are strong arguments to support these hopes. However, some industrial experiments gave bad results, which led many practitioners to have some doubts about the actual practicability of formal techniques. In this context, one of our industrial partners — an avionics manufacturer with the need of highly dependable embedded software — asked us to study whether it is of interest to introduce some formal verification techniques in his development process. We agreed with our partner to start the project with some experiments on a typical real-life example. The example he selected is an embedded programmed circuit. We were given a description board, a C code and a safety property. This property was assumed to be verified.

The first step of the study was therefore to select the mathematical and computer tools to be used. The computer tools had to be widely distributed, preferably already used in avionics applications and sufficiently user-friendly to be handled by engineers in a production context. This latter requirement means that engineers do not have much time to make them confident with the tools. From a survey of the literature and a preliminary study of the circuit, we selected SMV [1] and PVS [2]. This choice is motivated by several reasons. First, both tools

meet at least the first two of the above requirements. Second, they are representative of their category — model checkers for SMV and proof assistants for PVS. Third, they had been coupled for the very interesting experiments conducted at Rockwell-Collins during the development of the AAMP-5 embedded processor [3, 4]. These latter experiments are not the only ones reported in the literature (see for instance [5]). However, they were of special interest for our partner who is involved in the development of the same kind of embedded systems.

Without skipping straight to the conclusion, we would like to forewarn the reader who is mainly interested in experiments with proof assistants that this article focuses on the use of SMV … in a PVS style. After some experiments, we decided actually to postpone a full study of the circuit with PVS. There are mainly two reasons for that. The first one is inherent to proof assistants: as noted in Reference 2, a tool such as PVS requires a quite long training period for one to be confident with. Therefore, in spite of the great interest of this tool, it is pretty clear that it cannot be used in our partner's industrial context. The second reason is more specific to the circuit itself. The available descriptions of the circuit stand at a low level. For confidentiality reasons, high-level specifications of the circuit were not available. Therefore, we were unable to do a full reverse engineering to go back to these specifications from which should normally start a PVS study. Moreover, a model checker was well suited and almost sufficient for the purpose of the study. PVS had however a great influence on our work. The methodology we adopted — we call it “navigate to convince” — is derived from the PVS philosophy.

The aim of this article is to report the experiments we performed, to discuss methodological issues, to provide some ideas to improve the tools and finally to draw some general

conclusions about the question we were asked. Namely, are formal verification techniques of interest in our partner's strongly constrained context?

The following sections describe the circuit, the mathematical and computer tools we used, and discuss methodological issues. Examples taken from the study illustrate the paradigm "navigate to convince".

A realistic example

The circuit that was the subject of this study presents most of the features of circuitry in the field: Boolean and analogic components, memory registers and feedback paths. This circuit is described hereafter (Fig. 1) as a graphical board. Inputs are at the left of the figure. Outputs are at the right. The C code that implements the circuit was also available. This code cannot be given here because it is too large and for confidentiality reasons. However, the graphical board provides a good insight about what the studied circuit looks like.

The informal property to validate was that*, "at any time and whatever are the values of the inputs, at most one of the outputs is set to 1"*.

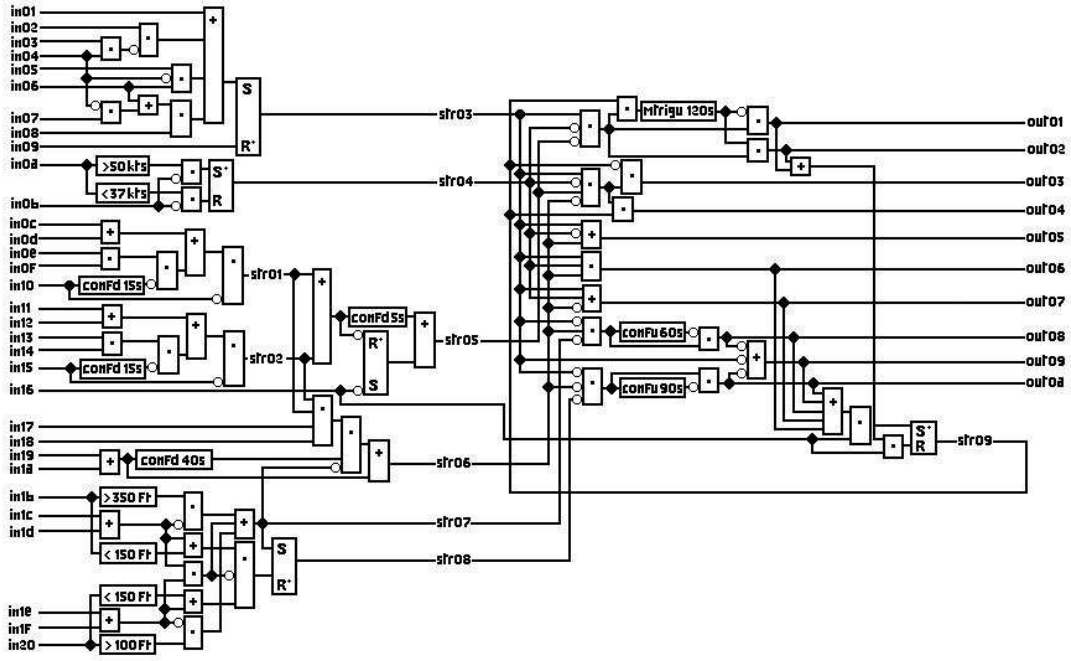

Fig.1. Graphical board of the circuit.

The circuit includes thirty-two inputs, numbered in01 through in20. Twenty-nine are Boolean and three are integers. There are ten Boolean outputs, numbered out01 through out0a and computed by seventy electronic components. These components are of the following types.

- Logical operators (`OR`, `AND`, `NOT`), that are denoted on the board respectively by +, -, and o.
- `RS` triggers (`R*S, RS*`), that are denoted on the board by these letters.
- Delay components (`Confd`, `Confu`, `Mtrigu`); the corresponding boxes on the board are labeled with the number of seconds to delay.
- Integral comparators, whose input is an integer and output is a Boolean; the sign of the comparison (“<” or “>”) and the reference value are indicated in the box.

The circuit is therefore not too large, but sufficiently complex to make pure simulation either far too costly or too incomplete, or both.

Tools

In order to be able to proceed to any verification, it is necessary to have a formal system description that ignores irrelevant details. So-called abstraction is a mean to reduce the information to handle (to achieve feasibility) and to focus on relevant features (for the efficiency of the process). Mealy machines are a well suited mathematical model to abstract circuits [6]. The temporal logic CTL is well suited to express safety properties [7]. SMV and PVS handle these formalisms.

Mealy machines

A Mealy machine is an input/output automaton (see, for instance, Reference 8 for a formal definition). Informally, a Mealy machine is composed of a set I of Boolean inputs, a set O of Boolean outputs, and a set R of Boolean registers. At any time, depending of the values of inputs and registers the machine computes the values $\omega(R,I)$ of the outputs and $\delta(R,I)$ of the registers at next time. This process is graphically illustrated on Fig. 2.

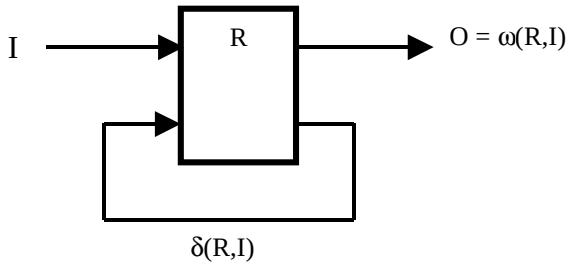

Fig.2. Mealy machines

Registers are used to represent the state of the components of the circuit, i.e. memories, triggers, delay elements, counters, loop back values (str09).

A Mealy machine describes implicitly a set of reachable states: starting from an initial state, this set is built stepwise by feeding the machine with all possible inputs and adding the new

reached states to the set until no more state is added (the fixpoint is reached). Behavioral properties of the system under study may therefore be expressed as properties of the graph of reachable states. Roughly speaking, the circuit contains 5 registers and 7 delay elements. Each register may be in two states. A good abstraction for delay elements is made of 16 states each. Therefore, the graph of reachable states may contain up to 2^{140} states! Indeed, this approximation does not take into account dependencies. The actual number of reachable states is very difficult to estimate. It is anyway huge, even by orders of magnitude smaller than 2^{140}. Moreover, this estimation does not take into account the size of the transition relation, which depends not only on the source and target state, but also on the possible inputs (here there are 2^{32} such inputs). To provide the reader with an idea of its size, it may be said already that the binary decision diagram encoding (an abstraction of) the set of reachable states does not fit into a half-gigabyte computer memory.

Computation Tree Logic

CTL formulae characterize sets of the reachable states. These Boolean properties refer to the "future" of a given state (past time operators are not allowed). Tense operators *F*, *G*, *U* or *X* are associated with quantifiers *A* or *E*, in order to express properties of "all possible futures" or "at least one possible future". More formally,

- Every atomic proposition (value of an input, an output or a register of the Mealy machine) is a CTL formula, and
- If f and g are CTL formulae, then so are $f \wedge g$, $\neg f$, AX f, EX f, A(f U g), E(f U g).

The other operators are derived from the above ones, according to the following rules:

$$f \vee g = \neg(\neg f \wedge \neg g)$$

$$AF\ g = A(true\ U\ g)$$

$$EF\ g = E(true\ U\ g)$$

$$AG\ f = \neg E(true\ U\ \neg f)$$

$$EG\ f = \neg A(true\ U\ \neg f)$$

A state satisfies an atomic proposition if this atomic proposition is true in the state. The semantics of connectives $\neg$ and $\wedge$ is as usual. A state satisfies the formula $AX\ f$ (resp. $EX\ f$) if all (resp. one or more) of its successors satisfy f. Finally, a state satisfies the formula $A(f\ U\ g)$ (resp. $E(f\ U\ g)$) if all (resp. one or more) of the paths starting from the state are such that all the states satisfy f until a state satisfying g is encountered. The semantics of CTL operators is illustrated graphically on Fig. 3 (where states that satisfy f are filled with black and those that satisfy g are filled with grey).

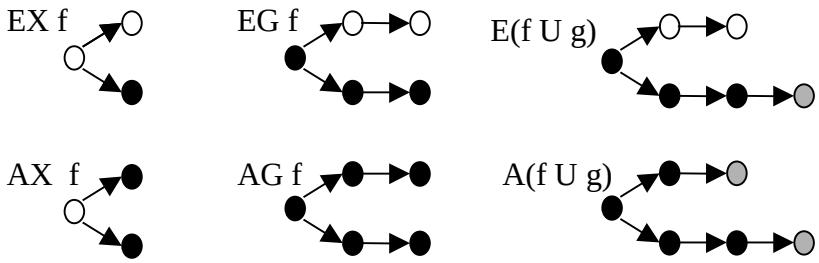

Fig. 3. Validity of CTL formulae

A verification consists in proving that a given assertion is true in the initial state, or in exhibiting a subset of reachable states (including the initial one) in which it is false. It is worth noticing that different tools, based on various technologies, can be used to perform such a verification. The Mealy machines/CTL formalism is versatile.

The SMV system

K. McMillan has designed the SMV (Symbolic Model Verifier) system in 1992 at Carnegie-Mellon University [1]. The SMV input language is designed for the description of sequential

circuits, including Boolean, ranged integer, and enumerated variables and built up from parameterized modules. Both synchronous and asynchronous systems can be described. Partially designed, abstract and non-deterministic behaviors are allowed. Hence, Mealy machines are easy to specify.

Once the modules are defined, the MAIN module describes the whole system by composing the different (sub) modules. The specification to verify is expressed, in the MAIN module, as a CTL property of initial state. Without any action from the user, SMV determines whether the specified formula is true. SMV carries out two kinds of answers:

- *The specification is verified*;
- *The specification is not verified*, in which case SMV exhibits one counter-example.

While using SMV along the study, it appeared that the diagnosis interface is not informative enough: exhibiting a single counter-example is too little an information to understand and to characterize the situations in which the property under study is falsified. This point will be discussed later.

The strength of SMV comes from its use of a Binary Decision Diagram (BDD) technology [9, 10] to encode sets of states (hence the prefix symbolic). Huge sets of states may be encoded as demonstrated in [11]. We do not present here BDDs, the interested reader should see the cited papers or Reference 1.

Aralia

In order to get more informative answers than those given by SMV, we used Aralia [12]. Aralia is dedicated to the assessment of Boolean reliability models such as fault trees or events trees (see for instance [13] for an introduction to risk assessment techniques). It is developed, since 1994, by one of the authors as a part of a partnership with several large

companies. It is now commercially distributed. As SMV, Aralia relies on a BDD representation of Boolean functions. It includes many features to perform both qualitative and quantitative (i.e. probabilistic) analyses of the models. It is now used as the reference tool for probabilistic safety studies of French nuclear installations. For the present study, we used only two features of Aralia: the possibility to deal with quantified Boolean formulae and the computation of prime implicants. In the context of reliability studies, prime implicants can be interpreted as minimal sets of basic component failures that induce a failure of the whole system [12]. For what concerns circuit analysis, the use of Aralia could (should!?) be efficiently replaced by extending SMV with the two mentioned features.

Discussion

As any choice, the one we made to use SMV and Aralia was partly driven by objective reasons and partly due to purely subjective ones. Among the objective reasons, the industrial context, as already explained in the introduction, plays the main role. It is worth mentioning however that both authors have quite a good background in the use of model-checkers for they were involved in the development of such tools. In our opinion, model-checkers correspond to engineers' way of working. They make it possible to design models and to play with these models, i.e. to perform computations. From that point of view, they are the logical counterparts of numerical calculus tools. SMV was particularly well suited for the present study because it has been designed to deal with sequential circuits. It is therefore very easy to describe such a circuit in the SMV input language and to verify properties (as soon as the CTL basic mechanisms are understood). Nevertheless, other model-checkers, such as Spin [14], could be considered as well. Another interesting alternative could be also to write a specification of the circuit using a reactive language such as Lustre [15]. However, the model

checker associated with Lustre (Lesar) is not as powerful as SMV. The interest of proof assistants such as PVS was already discussed in the introduction.

Methodological issues

A classical exponential blow up

We began the work by (manually) translating the C code of the whole circuit into a SMV description of a Mealy machine. This was achieved relatively quickly. Then, we expressed the property to be verified in CTL, which is also quite easy. However, the verification of the property led to a disastrous run out of memory, even on the very large memory computer we use in such occasions (a half-gigabyte). The various abstractions/reductions of the range of admissible values for numerical variables such as triggers were not sufficient to overcome this exponential blow up. Nor was the use of a partitioned transition relation [16]. Therefore, it became clear that a rough use of SMV would be not sufficient to reach the goal. This led us to a more deductive approach, mixing different concepts and tools. We do describe this methodology here and we shall illustrate it in the next section.

Navigate to convince

The following sentence, taken from Reference 2, illustrates the PVS philosophy: “to enforce conviction that a system meets its specifications, one can either test it by running on real data sets, or test it symbolically by proving lemmas and theorems on its executions”. Indeed, PVS is designed to follow the second line. In the case of our study, this approach consists in cutting the circuit into smaller units, then in proving relevant properties on these small units, and finally in proving compositional properties of the units. This is not actually a linear process, but an iterative one, embedding the three activities. Cutting is led by the need for realistic-

sized independent pieces and by the topology of the circuit. Properties to assert on the pieces or their assembling are either derived from the global specification (in a top down way) or from general properties on the pieces themselves (in a bottom up way).

Let us remark that the nature of the tool that is used to prove properties is not very important. The process is robust in the way that a model checker and/or a proof assistant may be used alternatively. The most efficient tool depends of the property to assert: in an exploratory phase, or to prove a combinatorial property, a model checker should be better suited. To prove the validity of an abstraction, a proof assistant should be of a better use.

We call “navigate-to-convince” the methodology that consists in analyzing pieces of the system in both top down and bottom up ways.

Divide to conquer

Real-life applications are in general too large to be verified in a straightforward way. Therefore, they have to be decomposed. What is the decomposition process and whether it can be, at least partly, automated depend on the application and … the verification tools, see for instance references [17,18,19]. In the case of the study presented in article, the decomposition was processed manually. It would be hard to give a formal algorithm to do it fully automatically. However, we used a number of rules of thumb:

- We tried to make parts not too large (to be able to prove lemmas) but not too small (to make the lemmas of interest). Parts with more than ten components are in general too large. Parts with less than three components are hardly of interest.
- We tried to isolate the parts that are independent from the remaining of the circuit. As we shall see, lemmas can be easily proved about these parts. These lemmas make explicit the relationship between the inputs and the outputs of the parts.

- We tried to isolate complex components such as delay elements, triggers, … Several models can be adopted for these components. It is therefore of interest to consider, *mutatis mutandis*, the impact of a change of abstraction level for one of them.

It would be unrealistic to design a fully automated decomposition process. However, graphical tools and graph algorithms (for instance to look at isthmus) would be useful.

The translation C to SMV

The translations from the C code to SMV and to Aralia were performed manually. This is indeed a source of errors. It would be interesting (and much safer) to compile automatically the C code, or even better a higher-level specification of the circuit, into the input language of the model checker. This is precisely the main objective of reactive languages such as Lustre [15]. We did not develop a compiler because it would have been too costly just for the purpose of the present study. However, such an automatic translation seems to be achievable. The C code of the circuit is actually quite simple and made of macro-definitions that are plugged together. Combinatorial parts are easy to translate. The only hard point is therefore the translation of delay elements, triggers… A solution could be to design a library of translations for such components. For each complex component, translations at different abstraction levels could be proposed. In this way, the translation process would be semi-automated. The user would be in charge of selecting the abstraction levels and the system would be in charge of translating easy parts and to plug together the various components. Note finally that, in practice, the size of the corresponding SMV code is roughly linear in the size of the C code.

What are the relevant lemmas?

The main part of the analysis consists in proving general lemmas about the pieces of the circuit. These lemmas are driven either by the global property to be verified, or by general considerations about the possible behaviors of the pieces. A part of the circuit has in general some inputs and one (or few) output(s). It may also contain some registers (or delay elements). The lemmas that are expected to be of interest explicit the relationship between the inputs and the outputs. They typically answer the following questions:

- For which values of the inputs (and registers), the output is 1?
- Are there values of the inputs that enforce the output to be 1, whatever are the values of the registers?
- Are there values of the registers that enforce the output to be 1, whatever are the values of the inputs?
- Is it always possible to produce an output 1 in the future?

And so on, …

The need of counter-examples

When the tested property is not verified, SMV provides a single counter-example. In many practical cases, it appears that a single counter-example is not sufficient enough to understand why the property is not verified. More counter-examples are required (the need of counter-examples in the symbolic model checking framework was already discussed in [20]). Note that examples are also needed when the property is verified. Such examples help to understand why it is verified and what is necessary to verify it. To produce extra (counter-) examples, one has to remove the previous ones from the model. This is tedious work, and dangerous with respect to the model and the reliability of the process. In order to avoid this

drawback, we used another tool — Aralia — to compute the set of configurations falsifying the property. This set of configurations is obtained by computing the prime implicants of a, possibly quantified, Boolean formula extracted manually from the SMV description.

More formally, the problem of finding counter-examples can be set as follows. A Mealy machine is fully described by the two functions $\omega(R,I)$ and $\delta(R,I)$ that give respectively the value O of the outputs at time t and the value R' of the registers at time t+1 from the value R of the registers and I of the inputs at time t. Now, the generic question we are interested in can be formulated as follows. Is there a sequence of values I_1, …, I_n of inputs and R_1,…, R_n of registers such that (i) R_1 is the initial value of the registers (ii) $R_{t+1}=\delta(R_t,I_t)$ for t=1,…,n-1 and (iii) $O_n=\omega(R_n,I_n)$ satisfies (or falsifies) a given predicate q. The problem is indeed that n is unknown. The idea of Bounded Model Checking [21] is to unfold the machine up to a given n, which indeed works only if a small enough n does exist. We used another idea (which is new, as far as we know). It consists in looking for values I of the inputs such that $q(\omega(R,I))$ is satisfied whatever are the values R of the registers. We compute the prime implicants of the quantified Boolean formula $\lambda I \, \forall R \, q(\omega(R,I))$. This is indeed an lower approximation of the set of inputs I_n such that there exists a sequence $(I_1,R1)\ldots,(I_n,R_n)$ that verifies (i), (ii) and (iii). But, it is of a great interest during the navigate-to-convince process. Moreover, the computation does not depend on n.

Discussion

As already said, it would have been too costly to design such a compiler just for the purpose of the present study. This is *a fortiori* true for the design of a tool in charge of the decomposition of the circuit. Moreover, we are convinced that, as soon as we are dealing with large systems, a fully automated verification process is out of accessibility, because of

exponential blow up problems. This means that the models, possibly obtained via an automatic or a semi-automatic translation process, should be manually tuned. In other words, we have to deal with models that are abstractions of the system. It is doubtful that the human intervention can be removed from such a modeling/abstraction activity. This does not mean that nothing can be done to improve the verification process. The design of a methodology for the use of model checkers, which is one of the objectives of the present paper, could be a step in that direction.

Experimental issues

In this section, we illustrate the methodology “navigate-to-convince” defined in the previous section by means of examples from the circuit.

Divide to conquer

The first step of the study consists in dividing the circuit into pieces of tractable sizes. According to the principles discussed in the previous section, the circuit is rather naturally decomposed into three areas (from left to right); each of these is then decomposed into several other ones. This is illustrated on Fig. 4, where the different parts are colored in different gray levels.

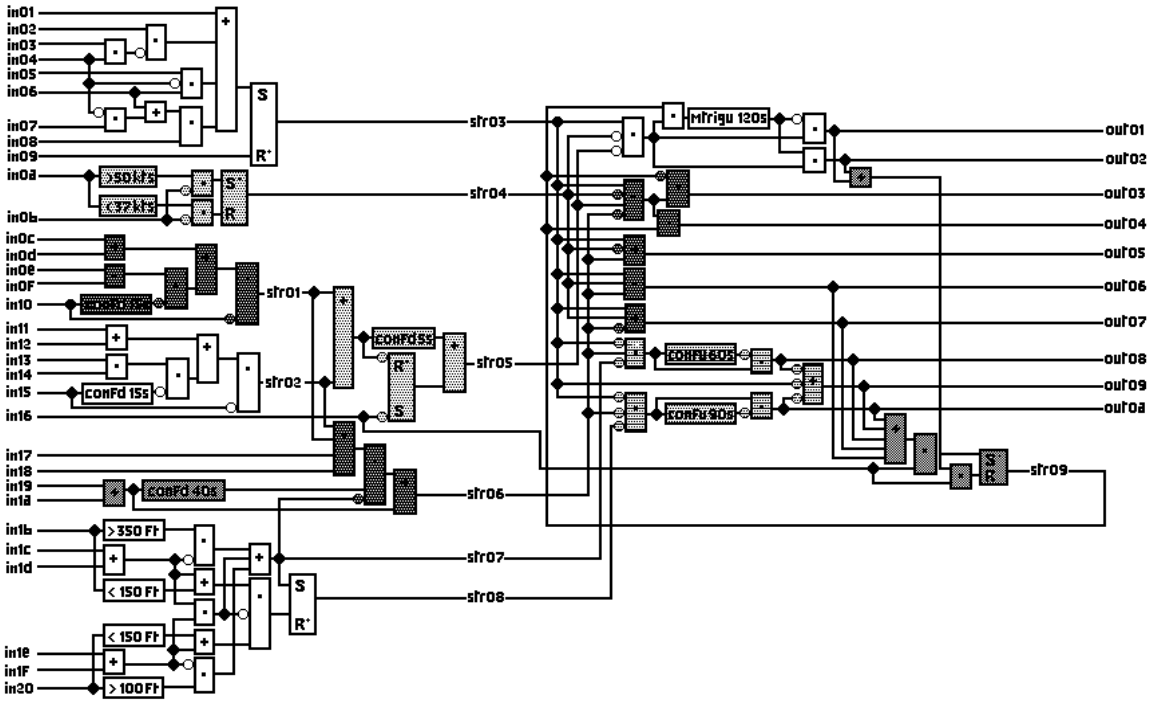

Fig. 4. Decomposition of the circuit.

Bottom up navigation

The circuit was decomposed into a two level hierarchy. Basic components of this hierarchy are good candidates for a bottom up analysis.

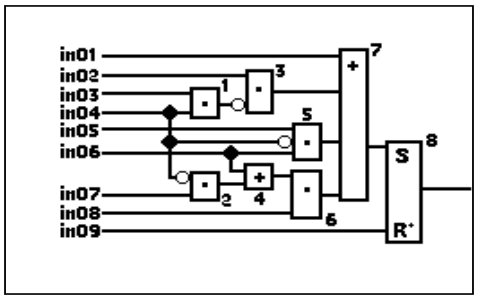

Fig. 5. The part La of the circuit.

Consider, for instance, the `La` area pictured Fig. 5. This component is isolated in the circuit (in terms of graphs, its output wire is an isthmus). An interesting property to assert is that "*whatever the state of the circuit, there exists always a sequence of values for the inputs*

producing output 1, and another sequence producing output 0". A SMV modeling can be used for this validation. The C code corresponding to `La` is as follows.

```
str03 = RSstar_op(&RS_str03,
                  in09,
                  OR4_op(in01,
                        AND2_op(in02,
                              NOT_op(AND2_op(in03, in04))),
                        AND3_op(in05,
                              NOT_op(in04), in06),
                        AND2_op(OR2_op( in06,
                                    AND2_op(NOT_op(in04),
                                          in07)), in08)));
```

Where operators `ORi_op` and `ANDj_op` are just macro-definitions for the corresponding combinatorial operations. The `RSstar_op` trigger is defined by following function.

```
TBoolean RSstar_op(TBoolean* RSstarQ, TBoolean R, TBoolean S){
  if(!S && R )
    *RSstarQ = FALSE;
  else
    if (S)
      *RSstarQ = TRUE;
  return *RSstarQ;
}
```

The SMV code to model this function is as follows.

```
MODULE RSstar(R,S,RSinit)
VAR
  memory : boolean ;
ASSIGN
  init(memory) := RSinit;
  next(memory) := output;
DEFINE
  output :=
    case
      (!S & R) : 0;
      S        : 1;
      1        : memory;
    esac;
```

The pointer to the `Tboolean` typed variable is modeled by a permanent Boolean variable (a register for the Mealy machine). The initial value of this register is passed as a parameter, and

will actually never be assigned to prove properties. The value after each clock tick and the value of the output are fixed by the “case” statement. The SMV code is as follows.

```
MODULE
sheet_partLa(RSinit,in01,in02,in03,in04,in05,in06,in07,in08,in09)
VAR
  RS : RSstar(in09,La7,RSinit);
DEFINE
  La1 := in03 & in04;
  La2 := !in04 & in07;
  La3 := in02 & !La1;
  La4 := in06 | La2;
  La5 := in05 & !in04 & in06;
  La6 := La4 & in08;
  La7 := in01 | La3 | La5 | La6;
  La8 := RS.output;
```

It is clear enough that the SMV translation follows not only the C code, but the drawing as well. Now, we write down the main SMV module and the property to verify *“at every time (AG) there exists a sequence of input values (EF) leading to a state where str03 is true and it exists a sequence of input values leading to a state in which str03 is false”.*

```
MODULE main
VAR
  RSLa8init : boolean;
  in01 : boolean;  in02 : boolean;  in03 : boolean;
  in04 : boolean;  in05 : boolean;  in06 : boolean;
  in07 : boolean;  in08 : boolean;  in09 : boolean;
  La   : sheet_partLa(
            RSLa8init,in01,in02,in03,in04,in05,in06,in07,in08,in09);
DEFINE
  str03 := La.La8;
SPEC
  AG ((EF str03) & (EF !str03))
```

The SMV answer is the following, which confirms the hypothesis.

```
-- specification AG (EF str03 & EF (!str03)) is true
resources used:
user time: 0.02 s, system time: 0.01 s
BDD nodes allocated: 284
Bytes allocated: 917504
BDD nodes representing transition relation: 15 + 5
```

We can then refine the property and wonder whether, from any reachable state, a long input sequence is needed to reach a state where str03 is set to 1 (resp. 0). A way to achieve this is to test sequences of length 1, then 2, and so on. In our example, there always exists an input value producing the required output, whatever the value of the register is. The SMV specification of this property consists in replacing in the previous the `EF's` by `EX`'s.

The use of Aralia

The next refinement is to enquire which inputs lead to the required output (e.g. str03 set to 1). Unfortunately, the actual release of SMV does not allow such a question. That is the reason to use Aralia. As Aralia is not designed to handle sequential circuits, we have first to abstract the circuit in a combinatorial circuit, leaving free the registers' values. Then, these values are universally quantified to obtain valid results whatever the event sequence before the considered state is. The Aralia modeling of the `La` area of the circuit is as follows.

```
La1 := (in03 & in04);
La2 := (-in04 & in07);
La3 := (in02 & -La1);
La4 := (in06 | La2);
La5 := (in05 & -in04 & in06);
La6 := (La4 & in08);
La7 := (in01 | La3 | La5 | La6);
La8 := (La7 | (-in09 & La8.memory));
str03-true  := forall La8.memory  La8 ;
str03-false := forall La8.memory -La8 ;
```

We can then compute the prime implicants of the functions associated to names `str03-true` and `str03-false`. For `str03-true` we obtain following prime implicants.

```
{in01}
{in02, -in03}
{in02, -in04}
{-in04, in05, in06}
{-in04, in07, in08}
{in06, in08}
```

This can be read as follows. "*if* `in01` *is set to 1, then whatever the rest of it,* `str03` *is set to 1*". We notice that this result can be confirmed by running the C code or testing the SMV modeling with the specification `AG (in01 -> str03)`.

This example illustrates the interest of prime implicants for the verification process: they give a concise view of the reasons why a property is verified or not. In order to provide the user with clear examples, model-checkers should embed a module to compute and to display prime implicants.

Models for complex elements

While modeling the other parts in the lower area of the circuit, we encounter complex (i.e. involving memory) operators. Modeling the `RstarS` operator is similar to the `Rsstar` operator; we have to deal with comparators such as input<value or input>value, and the `Confu`, `Confd` and `Mtrigu` boxes.

Modeling comparators is an easy job. Their inputs are modeled in SMV by three valued variables to avoid the combinatorial explosion. For example, the input of comparators `>50kts` and `<37kts` are modeled by means of a variable of domain {0, 40, 80}. It is not difficult then to translate this variable into Boolean variables to be used by Aralia.

Modeling the operators `Confd` and `Confu` is more delicate. These operators update a counter according to their input values. For example, the C code for the operator `Confd` is as follows.

```
TBoolean Confd_op(TConfSymbol* PConf,
                  TBoolean  BE,
                  TBoolean  BIC,
                  int p,
                  int m) {

 TBoolean BS;

 if (BIC && !BE)
   PConf->Count = p-m;
```

```
  if (BE) {
    BS = TRUE;
    PConf->Count =1;
  }
  else
    if (( 0 < PConf->Count) && ( PConf->Count <= p)) {
      PConf->Count ++;
      BS = TRUE;
    }
    else {
      PConf->Count = 0;
      BS = FALSE;
    }
  return BS;
}
```

The behavior of the operator depends not only from the input `BE`, but also from the initial values `BIC`, `p` and `m`. In order to model it in SMV, we choose domains for the variables depending of the constants in the C code (`p`=10, `m`=4 or 5). The SMV code is somewhat different from the C code, but the correspondence is easily found.

```
MODULE Confd(BE, BIC, p, m, initCount)
VAR
  count : {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
ASSIGN
  init(count) := initCount;
  next(count) := c2;
DEFINE
  c1 := case
        (BIC & !BE) : p - m;
        1           : count;
      esac;
  c2 := case
        BE              : 1;
        ((0<c1)&(c1<=p)) : c1+1;
        1               : 0;
      esac;
  BS := case
        BE              : 1;
        ((0<c1)&(c1<=p)) : 1;
        1               : 0;
      esac;
  output := BS;
```

An interesting abstraction, that enables computations with Aralia, is to extend the idea already used to model triggers: what is merely combinatorial is expressed, and free variables are created for the registers. As an illustration, we can model the fact that the BS output is set to 1 if the input BE equals 1 and takes an undetermined value otherwise. Aralia equations to model the Le area are therefore as follows.

```
Le1 := (in19 | in1a);
Le2 := (Le1 | Le2.nonDeterminism);
```

The Mtrigu operator is modeled in a similar way in SMV. On the Aralia side, no merely combinatorial part can be exhibited. Its output is thus considered fully non-deterministic.

Top down navigation

Up to now, we showed examples of bottom-up navigation. Let us show now an example of top-down navigation.

The property "*one and only one output set to 1*" implies that for all subsets of outputs, at most one of them can be set to 1 (all outputs of the subset may be set to 0). Therefore, we can isolate the sub-areas Ha and Hb of the circuit and wonder whether there exists a value of inputs of this sub-circuit that falsifies this derived property. The Aralia model for these sub-areas Ha and Hb is the following equations set:

```
Ha1 := (str03 & -str04 & -str05);
Ha2 := (str09 & Ha1);
Ha3 := Ha3.nonDeterminism;
Ha4 := (-Ha3 & Ha1);
Ha5 := (Ha3 & Ha1);
```

```
Hb1 := (str03 & -str04 & str05 & -str06);
Hb2 := (str03 & -str04 & str06);
Hb3 := (str03 & str04 & str06);
Hb4 := (str03 & str04 & -str06);
Hb5 := (-str09 & Hb1);
Hb6 := (str09 & Hb1);
```

One can then express the derived property and compute the prime implicants of it, quantifying universally the non-deterministic output value of operator `Mtrigu.`

```
constraint := #(2,7,[out01,out02,out03,out04,out05,out06,out07]);
property := forall Ha3.nonDeterminism constraint;
```

One gets only one prime implicant: {`str03`, `-str04`, `-str05`, `str06`}. This leads to another property to verify: the values 1, 0, 0 and 1 must never occur simultaneously on wires `str03`, `str04`, `str05` and `str06`. We have there an example of "proof obligation" as quoted in PVS. Notice that this lemma was not easy to produce, and the help from Aralia has been invaluable.

Going on in this process, we come to the conclusion that there are some input values falsifying the required property.

> **Under the condition that circuit inputs are pair wisely independent, the property "*at any time one and only one output set to 1*" is false.**

This was quite surprising for us since our partner claimed that the property is verified.

In order to exhibit sets of input values falsifying the specification, it has been sufficient to connect together the Aralia models of the different parts of the circuit. For example, the following set of values shows that the property can be falsified in one step (the other inputs may take any value).

```
in01=1, in0a<37kts, in0b=0, in0c=0, in0d=0, in0e=0, in11=0, in12=0, in13=0,
in19=1
```

This can be verified either by running the circuit, or using the SMV model. In the later case, we have to fix the input values, otherwise the combinatorial explosion would prevent from computing the set of states.

Hence, by means of an iterative process, using bottom up and top down approaches in a balancing way, using Aralia and SMV alternatively, we have been able to determine rather quickly that the circuit did not meet its specification.

Conclusion

In this article, we reported a real-life application of so-called “Formal Methods”. The part of the project we were involved in was to verify that an embedded programmed circuit satisfies a given safety property. By means of the two tools SMV and Aralia and a methodology derived from the PVS philosophy, we were able to show that the circuit does not meet its specification, conversely to what was claimed by our industrial partner.

From these experiments, we may draw a number of general conclusions.

The methodology we adopted, so-called “navigate to convince”, seems to be well suited to deal with the kind of applications we were involved in. This approach summarized by the three steps: (1) decomposition of the system into pieces, (2) proof of combinatorial lemmas on the pieces, (3) collection of the lemmas in order to prove or disprove the required property.

- The decomposition of the circuit by “topologically” cutting it into “relatively independent” pieces of “tractable sizes” seems arbitrary and intuitive. It is actually a reverse engineering technique, and therefore essentially a manual process. It could be the case however that some graph techniques provide a useful help to do it.
- The proof of the properties on the pieces, using a model-checker such as SMV, augmented by a prime implicant engine (e.g. Aralia) is the cornerstone of the whole process. We discussed it in details in this article.

- The collection of the many small lemmas proved about pieces has to be done very carefully. It is probably the case that, for larger and more complex examples than the one presented here, a proof assistant such as PVS provides a useful help to do so.

It should be noticed that the above process is not a linear one. It involves both bottom-up and top-down navigation through the system under study and the properties to be verified.

SMV is a very efficient tool when used as an interactive logical calculator. However, it appears clearly that it lacks of functionalities to provide the user with illustrative answers. In order to overcome this lack, we used Aralia. The use of this later tool could be replaced advantageously by extending SMV with a new module to produce sequences, possibly relying on new principles.

Finally, we would like to say that we are convinced that formal verification techniques can be inserted successfully in the development process, even under the strong conditions set by our industrial partner. The methodology we adopted does not change radically the work of engineers. On one hand, it provides a general framework, i.e. some guidelines to manage the studies; on the other hand, it is pretty easy to be confident with a tool such as SMV, the functionalities of which may be learned stepwise, when needed.

References

[1] K.L McMillan. *Symbolic Model Checking*. Kluwer Ac. Pub. 1993.

[2] J.Crow, S.Owre, J.Rushby, N.Shankar, and M.Srivas. A Tutorial Introduction to PVS. In *Proceedings of the Workshop on Industrial Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995. Also technical report ftp available at http://www.csl.sri.com/sri-csl-fm.html.

[3] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model checking with automated proof checking. In Pierre Wolper, editor, *Computer Aided Verification*, CAV'95, LNCS 939, pages 84-97.

[4] N. Shankar. PVS : Combining Specification, Proof Checking, and Model Checking, In *Proceedings of FMCAD'96*, LNCS 1166.

[5] A.Th. Eirìkson and K.L. McMillan. Using Formal Verification Methods on the Critical Path in System Design: A Case Study. In *Proceedings of Computed Aided Verification, CAV'95*. P. Wolper Ed. LCNS 939. 1995.

[6] G. Barett and A. McIsaac. Model checking in a microprocessor design project. In O. Grumberg, editor, *Lecture Notes in Computer Science*, vol. 1254, 214-225. CAV'97, Springer, 1997.

[7] E.A Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B : Formal Models and Semantics, 997-1072, Elsevier Sc. Pub., 1990.

[8] J.E Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. 1979.

[9] R. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677-691, August 1986.

[10] K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE 0738, 1990.

[11] J.R. Burch, E.M. Clark, K.L. McMillan, D.L. Dill, and L.J. Hang. Symbolic model checking: 10^{20} states and beyond. In *Symposium on Logic in Computer Science*, 1990.

[12] Y. Dutuit and A. Rauzy. Exact and Truncated Computations of Prime Implicants of Coherent and non-Coherent Fault Trees within Aralia. *Reliability Engineering and System Safety*, 58:127-144, 1997.

[13] J.D. Andrews and T.R. Moss. *Reliability and Risk Assessment*. Longman Scientific and Technical. ISBN 0-582-09615-4. 1993.

[14] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series.1991. ISBN 0-13-539925-4.

[15] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, The synchronous dataflow programming language Lustre, *Proceedings of the IEEE*, 79(9),pp 1305-1320, 1991.

[16] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic Model Checking with Partioned Transition Relations. In Proceedings of the 1991 International Conference onVLSI. August 1991.

[17] J. Lind-Nielsen, H.R. Andersen, G. Behrman, H. Hulgaard, K. Kristoffersen, and K. Larsen. Verification of large state/event systems using compositionnality and dependency analysis. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems, TACAS'98*, volume 1384 of *LNCS*, pages 201-216, 1998.

[18] G. Behrman, K. Larsen, H.R. Andersen, H. Hulgaard, and J. Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionnality and dependency analysis. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems, TACAS'99*, volume 1579 of *LNCS*, pages 201-216, 1999.

[19] W. Lee, A. Pardo, J.Y. Jang, G. Hatchel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *Proceedings of the IEEE/ACM International*

Conference on Computer-Aided Design, CAD'96, pages 76-81, IEEE Computer Society Press, November 1996.

[20] R. Hojati, R.K. Brayton, and R.P. Kurshan. BDD-based debugging of designs using language containment and fair CTL. In C. Courcoubetis, editor, *Proceedings of the Conference on Computer Aided Verification, CAV'93*, volume 697, pages 41-58. *LNCS*, 1993.

[21] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, vol. 1579 of *Lecture Notes in Computer Science*. Springer Verlag. 1999.