Verification of Safety-Critical Systems: A Case Study Report on Using Modern Model Checking Tools

Antti Jääskeläinen[1], Mika Katara[1], Shmuel Katz[2], and Heikki Virtanen[1]

1 **Department of Software Systems, Tampere University of Technology**
PO Box 553, 33101 Tampere, Finland
{antti.m.jaaskelainen,mika.katara,heikki.virtanen}@tut.fi

2 **Department of Computer Science**
Technion – Israel Institute of Technology
Haifa 32000, Israel
katz@cs.technion.ac.il

Abstract

Formal methods are making their way into the development of safety-critical systems. In this paper, we describe a case study where a simple 2oo3 voting scheme for a shutdown system was verified using two bounded model checking tools, CBMC and EBMC. The system represents Systematic Capability level 3 according to IEC 61508 ed2.0. The verification process was based on requirements and pseudo code, and involved verifying C and Verilog code implementing the pseudo code. The results suggest that the tools were suitable for the task, but require considerable training to reach productive use for code embedded in industrial equipment. We also identified some issues in the development process that could be streamlined with the use of more formal verification methods. Towards the end of the paper, we discuss the issues we found and how to address them in a practical setting.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Functional safety, SIL-3, model checking, tools

Digital Object Identifier 10.4230/OASIcs.SSV.2011.44

1 Introduction

Companies developing safety-critical systems must balance between safety requirements imposed by standards and productivity requirements. On the one hand, the higher the safety integrity requirements, the more time and effort are needed for validation and verification activities. On the other hand, companies producing less safety-critical systems often face fierce competition and are required to put more emphasis on the overall efficiency of the development process.

Certification is another driving force in the field. Many companies are trying to get their products certified in order to help marketing efforts. The new machinery directive in the EU, for instance, is still based on self-declaration in the case of most type of machines; the manufacturer labels the product with the "CE" marking without formal type examination. However, certification by an independent assessment organization may still be required by customers and/or for marketing reasons. It is also seen as an important step if an accident should occur and investigation of the development practices takes place.

6th International Workshop on Systems Software Verification (SSV'11).
Editors: Jörg Brauer, Marco Roveri, Hendrik Tews; pp. 44–56

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

IEC 61508 [15] is a basic standard on functional safety; a new edition 2 of the standard was released in April 2010. The standard classifies safety-critical systems into four Safety Integrity Levels (SILs), SIL 4 corresponding to the most critical and SIL 1 the least critical type of system. The standard presents methods used for the verification and validation of safety-critical hardware and software. For each SIL level, there is a set of Highly Recommended, Recommended and Not Recommended methods. In addition, for the use of some methods the standard does not indicate any recommendation on certain SIL levels.

Systems can be composed of elements and subsystems having a predetermined Systematic Capability (SC) on the scale 1-4 corresponding to the SIL level of the whole system. For example, SIL 3 level systems can be composed of elements having SC 3 or 4 when used according to the instructions given in the elements' safety manuals. IEC 61508 is not harmonized, i.e. it does not fulfill the requirements of the European directives as such, but is often referred to by other, harmonized, standards (such as EN ISO 13849-1 and EN 62061) in relation to requirements imposed on the development of safety-critical systems.

Formal methods are considered an important technology in the development of safety-critical systems. While the scalability and usability of tools still pose challenges in the development of non-safety-critical systems, safety-critical systems are somewhat different in this respect. On higher SIL/SC levels there are fewer productivity constraints and perhaps more time to learn new techniques that can help in validation and verification efforts. Moreover, safety-critical systems should be kept rather simple in order to limit the needed verification and validation activities. Thus, in spite of the scalability problems, it it often feasible to prove correct at least some parts of the system using formal methods. Moreover, formal methods are well represented in the IEC 61508 standard for developing high SIL level systems. They can also be used on lower SIL levels to replace some less formal techniques, such as certain types of testing.

Nevertheless, there have been major impediments in using formal methods. Performance of the old tools and the computing power available was too limited in order to solve real life problems. Moreover, special expertise was required to use the tools. Nowadays, there is evidence in the literature that new tools can solve practical problems given the increased computing resources available. Unfortunately, however, there is still lack of user experience reports that would discuss the required expertise to use the modern tools.

Towards these ends, we describe a case study where we experimented with a formal verification technology in an industrial case study. The case study subject was a simple 2oo3 (2-out-of-3) voting scheme used for redundancy in a SC 3 level shutdown system. The system development is being done according to the IEC 61508 standard and certification is being conducted by an independent organization.

In the case study we treat a typical industrial development where pseudo-code (or programming language code in, e.g., C) is first written and shown correct relative to requirements of the module under development. This is then handed off to a separate development team as the basis of a hardware or firmware design in a low-level hardware design language such as Verilog or VHDL. This too must be shown correct relative to (often more detailed) requirements. The main practical tasks are (1) to ensure that the high- and low-level properties checked actually express the needed requirements and are easy to write, (2) to facilitate verifying the pseudo-code level relative to the properties, and (3) to similarly show that the chip design satisfies the needed properties, while showing consistency with the upper level solution.

For verification we used two bounded model checking tools, CBMC and EBMC [6]. Model checking as a technology does not require as high a level of expertise as, for instance, theorem

proving. Moreover, these tools were easily available and supported the input formats we were able to work with. In addition, they support the existing development process and no major changes in the work flow are required.

While the standard does not require the use of formal verification in the case of this particular system, formal verification can complement less formal verification methods, such as testing and simulation, and somewhat ease the certification process. Moreover, if the development process could be changed in the future to better take advantage of the formal verification technology, some of the less formal techniques could possibly be replaced with it.

Since the verified system is very simple, the focus of this paper is on reporting experiences in using the verification tools in the particular industrial context of safety-critical systems development rather than in the verification technology itself. The results of the case study suggest that while suitable tools might be hard to find, together with the process changes, they could provide better evidence for the correctness of the system. Should it be possible to replace some informal techniques with more formal ones, productivity gains could also be achieved. Nevertheless, the efficient use of model checking tools requires expertise, so considerable training may be needed in order to equip the developers with the skills necessary to use such tools.

The structure of the paper is as follows: In Section 2 we present the background of the tools used in the case study and discuss related work on how formal methods and related tools are used in various tasks in software and hardware development. Section 3 introduces the case study. Due to confidentiality restrictions, some details of the shutdown system have been omitted. Finally, the lessons learned from the case study are discussed in Section 4.

2 Model Checking Safety-Critical Systems

In this section, we first introduce the basic concepts related to model checking in general and bounded model checking in particular. Then we move on to discuss related work on how formal methods and related tools are used in various tasks in software and hardware development.

2.1 Model Checking and Bounded Model Checking

Model checking [8] is a formal verification technique in which all possible execution paths of a model of a system or component are checked for a given property, where the model must have a finite number of possible states (although there can be infinite computations). The property to be checked is generally given in some form of temporal logic [18]. This allows expressing assertions about the final values of a module, invariants that should be true also at intermediate stages, as well as assertions about, e.g., responsiveness of a system to requests or stimuli. The model of the system is called a Kripke structure, and is a graph with nodes that each represent a state of the system, and directed edges where each represents an operation that moves the system from the source state to the target.

The main problem with model checking is that the number of states in a system can become unmanageably large. Thus model checking techniques are intended to overcome this difficulty. Among the classic approaches are representing the states symbolically in data structures known as binary decision diagrams (BDD's), and creating smaller "abstract" models that combine many states into one (so that if the smaller model is shown correct for a desired property, the original model is also guaranteed to be correct).

Model checking tools originally had their own notations for expressing the models, e.g., in the SMV model checking tool [7]. The tool and its notation were used either to show that

a design of a key algorithm was correct, or code was translated to the notation of the tool involved [11].

More recently, tools have been developed to directly take as input the code of the component to be checked (e.g., in C or Java), and to use *assert* statements to indicate at what point an assertion should be correct. In addition, the underlying technology of model checkers has changed: today it is common to translate both the model (or code) and the assertions to a complex boolean formula, and use a SAT (satisfiability) solver [19] or extended techniques called SMT [20] to determine whether the formula can be made true for some assignment of values to the variables in it. In fact, the formula constructed is equivalent to encoding the execution of the model, and asserting the *negation* of the property we want. Thus finding a set of values for the variables in the formula is equivalent to finding a counterexample for the property, because it represents a computation of the system that does not satisfy the desired property.

Both in order to create smaller models, and to ensure that any counterexample execution paths are as short as possible, *bounded* model checking has been used. In this approach, a bound is put on the length (number of states) of paths that will be checked. Thus for some n, all possible paths of length up to n are checked. If a counterexample is found, it can be analyzed to detect the bug. While if none exists for paths up to n, the bound can be increased, until a bound longer that any path in the program is reached, or the user decides that longer paths can be ignored.

Modules to ensure safety-critical properties of industrial software often regulate control or repeatedly test whether shut-down is necessary. Such modules are generally limited in their state-space, and each round of application is bounded in length. Thus bounded model checking is appropriate, and often can achieve full verification. Full model checkers, such as SATabs are appropriate for larger programs, but, as noted on the home webpage of that tool [23], can only automatically check for restricted properties such as array bounds, buffer overflows, or built-in exceptions, because of the needed abstraction step in going from code to a model.

In this work we show a case study where the computations are of a fixed length at each activation of the module investigated, so many of the more complex issues are irrelevant. We investigate whether tools for bounded model checking are sufficiently robust and user-friendly to be practically used to verify and increase the reliability of software or firmware embedded in industrial equipment.

In this case study, we used two bounded model checkers, namely CBMC and EBMC [6]. The former enables software model checking and supports ANSI-C and C++ as input languages. The tool performs verification by first unwinding the code loops and then passing the results in an equational form to a decision procedure (e.g., a SAT solver). In many cases, the tool can check that enough unwinding is performed, and thus the complete state space is considered in the analysis (sound verification). If the formula that encodes the program unwindings is satisfiable, i.e., contains an invalid program path, then the tool will produce a counterexample. There are also command-line options to limit the number of times the loops are unwound or the number of program steps to be processed, and to stop checking that enough unwinding is done; this allows using the tool for bug hunting in cases where no useful bound exists and properties cannot be proven correct. On the other hand, EBMC is a tool for hardware verification supporting input in Verilog and related formats. However, VHDL is not among the supported input formats. Both tools are available in binary format for Windows, Linux and MacOS.

2.2 Related Work

In the following, we describe some application examples of formal verification techniques in relation to software and hardware development. It is worth paying attention to the way model checking is used and what kind of impact it has for the development process and overall quality.

Björkman et al. [4] verified stepwise shutdown logic in the nuclear domain and used model checking in the traditional way: the design was converted to a dedicated verification model and the requirements in the specification were translated into logical formulae. They used a model checking tool for proving that the verification model satisfies the formulae. Obviously, this use of model checking is rather demanding and laborious because of the model transformations needed, but it has some advantages as well. Already while constructing formal models, many omissions and contradictions become clearly visible, and larger systems can be verified because irrelevant details can be omitted from the abstract verification model.

The cited experiment shows the most valuable benefit of formal methods too. Because all of the modeled behavior is fully covered, no issue can hide itself in the verification model. However, the proof is valid only if the abstract verification model corresponds to the design and the formulae cover all of the requirements. One of today's research challenge is to find new ways of applying formal methods so that the artifacts used in proofs would be more closely related to the specification, design and implementation languages used in mainstream software and hardware development; this would reduce the need for error-prone manual transformations.

Even though formal methods may not be applicable always as such, they can still be helpful. For example, testing can benefit from their use. Angeletti et al. [1] reported an experiment in the railway domain where bounded model checking was used to semi-automatically generate test cases in order to gain full coverage requested by the EN 50128 guidelines for the software development of safety-critical systems at SIL 4 level.

In the experiment, the C code was augmented with failing assertions and the CBMC tool was used to compute the values of input parameters for each assertion to be reached. Obviously, the mere values of input parameters are not enough for defining test cases; in order to be useful, the test case must contain checks against the expected outputs. In our case study, such checks were encoded directly as conditions in assertions and verified on the fly. Unfortunately, the paper by Angeletti et al. does not state directly how the expected values were obtained and why on-the-fly verification was not used. A system of a few thousand lines of C code may be too large to be model checked, so the approach we used in our case study may not have been applicable as such, and unlike in our approach, test cases can be used to verify and validate the SUT in binary form without the source code.

In theory, any model having operational semantics can be verified by means of model checking and state transition systems can be used to model many other aspects of the systems than behavior in normal conditions. For example, there is a special Statecharts variant called Safecharts for modeling safety issues and their relations to functional properties [9].

In Safecharts there are special states for normal and defunct states for the components of the system and transitions between them. Events associated with those transitions model the breakdowns and reparations of the components. When these special states and events are synchronized with the states and actions of the functional layer, the behavior of the system can be modeled and formally verified, not only in normal operation, but in those situations in which parts of the system do not function properly [12]. This is very useful if the system cannot reach a safe stable state without controlled operations. In aerospace and nuclear domains this requirement is obvious, but also in the case of complex and big

machines there might be a need to shutdown slowly to prevent further breakdowns.

In addition to facilitating testing, formal verification can significantly reduce the need for testing. Kaivola et al. [16] used formal verification as the primary validation vehicle for the execution cluster of the Intel Core i7 processor and dropped most of the usual RTL (Register Transfer Language) simulations and all coverage driven simulation validation. They concluded that verification required approximately the same amount of work as traditional pre-silicon testing. Although not zero, the number of bugs that escaped to silicon was lower than for any other cluster.

In addition to describing how formal verification could replace testing, Kaivola et al. sketch some prerequisites for verification to be applicable in practice. In contexts where model checking can replace simulation-based testing, it can be seen as a clever and effective way of conducting exhaustive simulation.

Nevertheless, even a company like Intel has taken quite some time to introduce formal verification into the development process. Most likely the story began in 1994 when the Pentium FDIV bug was found [22] and seven years later they reported that they had verified the Pentium 4 floating-point divider [17]. As a pioneer in the field, Intel has made enormous investments in formal verification and for others, effort is likely far more moderate. Still, it may take considerable effort to establish the confidence needed to be able to supercede existing verification methods with more formal ones. However, they can used to complement the existing ones and provide diversity when needed.

The systems in the examples discussed above have very high integrity requirements and two of them are also large from the verification point of view. For example, the execution cluster of i7 is responsible for the functional behavior of all of the more than 2700 distinct microinstructions. The majority of safety-critical systems are much smaller and may not have such high integrity requirements. Nonetheless, formal methods can be a feasible alternative for the quality assurance of those because small verification problems are not as laborious to solve as it is generally thought and even small systems can have peculiar and critical faults, which can be almost impossible to find by other means.

3 Case Study

We now present our case study on using model checking to verify a simple element in a safety-critical system. In more detail, the goal was to use model checking tools to verify the implementation of the 2oo3 voting scheme in a SC 3 level shutdown system. This voting scheme (also known as triple-modular redundancy) is very popular in safety-critical systems because it provides a good compromise between safety and availability. Since availability is an important factor in industrial systems, such compromises are often searched for.

There are three distinct modules which receive the same input (from one, two or three different sensors) and shutdown is started when at least two out of three modules suggest it. In this case, the design follows the idle current design, i.e. the output is active when there is no need to shutdown the system. When at least two out of three inputs are active, the output is also active. If only one or zero inputs are active or the power is lost, the output should indicate a need to start the shutdown procedure. In practice, each input is a Boolean value, one indicating a normal situation and zero indicating the need to shutdown the system. If two or three of the input values equal zero, the voter unit outputs value zero and the shutdown procedure begins. If only one input equals zero, the process can continue (with a possible log message indicating some potential problem in the corresponding module). Thus, the system is able to mask a fault in one of the modules, allowing the system to continue its

operation. The interested reader is referred to [24, p. 132] for more elaborate discussion on this voting scheme.

3.1 Working with the Pseudo Code

In this case the development process is such that the basic requirements are refined first and then translated into pseudo code. Typically, the pseudo code is augmented with a short textual description that may specify some basic properties of the solution depicted as pseudo code. The pseudo code is then implemented with a suitable concrete language; VHDL in case a programmable hardware solution is preferred. The tests for the implementation are derived from the requirements, which are managed in a requirements management tool.

The first stage of the case study was to verify the pseudo code. The tool used for formal verification was CBMC (version 3.9) and for that purpose the pseudo code was manually translated into C code. Since the implementation of the voting scheme with Boolean values is very simple, manual translation was considered adequate in this particular case. Moreover, because of the simplicity of the code, it was possible to derive eight test cases (2^3) that covered all possible input and output combinations.

The test cases were encoded as assertions in the C code and verified with the tool. This process also revealed that the property specified in conjunction with the associated pseudo code was somewhat vague and incomplete; the informal description didn't cover all the input/output combinations. We think that this represents a typical case of specifying simple designs: even though the requirements should be explicit and complete, it is very easy to ignore some details since the design is considered obvious.

The C code used with CBMC is listed in Figure 1. There are three parameters, corresponding to three inputs to the system; the `OCHY_Voter_State` variable is the output. The actual voting is implemented in the statement where `OCHY_Voter_State` gets assigned a value. The assertions corresponding to the eight test cases follow the assignment. The structure of the assertions was chosen to support understandability in the absence of the implication operator; another possibility would have been to use not and or operators to substitute for implication (and give the original form with the implication operator in a comment above the assertion, for instance). The current form also shows the locality of assertions in C.

3.2 Working with the VHDL Code

The second stage was to verify the actual implementation of the pseudo code in VHDL. Ideally, the verification tool should accept VHDL as input language, but for practical reasons we chose EBMC (version 4.1). Since EBMC uses Verilog as its input language, we first translated the VHDL code to Verilog using a VHDL to Verilog RTL translator tool [10] (version 2.0). The verification process was not as straightforward as in the case of the C code. We struggled with the syntax and the use of the tool since the information available with the installation package and on the tool website [6] was more limited than in case of CBMC. A significant practical difference with the tools was that the assertions were considered global in EBMC and local in CBMC. This made the reuse of assertions developed for the C code impossible.

Figure 2 shows how the voting is implemented in the Verilog code. The *always block* gets executed on the rising edge of either the clock or the reset signal. If the reset is active (zero), then the output is zero. Otherwise the voting occurs. Interestingly, the implementation

```
#include<assert.h>
void foo(int OCHY_comparator_state_ICH1,
         int OCHY_comparator_state_ICH2,
         int OCHY_comparator_state_ICH3) {
int OCHY_Voter_State = 0;

OCHY_Voter_State =
  (OCHY_comparator_state_ICH1 || OCHY_comparator_state_ICH2) &&
  (OCHY_comparator_state_ICH1 || OCHY_comparator_state_ICH3) &&
  (OCHY_comparator_state_ICH2 || OCHY_comparator_state_ICH3);

if ((OCHY_comparator_state_ICH1 == 1) && (OCHY_comparator_state_ICH2 == 1)
 && (OCHY_comparator_state_ICH3 == 1)) { assert(OCHY_Voter_State == 1);
}
if ((OCHY_comparator_state_ICH1 == 1) && (OCHY_comparator_state_ICH2 == 1)
 && (OCHY_comparator_state_ICH3 == 0)) { assert(OCHY_Voter_State == 1);
}
if ((OCHY_comparator_state_ICH1 == 1) && (OCHY_comparator_state_ICH2 == 0)
 && (OCHY_comparator_state_ICH3 == 1)) { assert(OCHY_Voter_State == 1);
}
if ((OCHY_comparator_state_ICH1 == 0) && (OCHY_comparator_state_ICH2 == 1)
 && (OCHY_comparator_state_ICH3 == 1)) { assert(OCHY_Voter_State == 1);
}
if ((OCHY_comparator_state_ICH1 == 1) && (OCHY_comparator_state_ICH2 == 0)
 && (OCHY_comparator_state_ICH3 == 0)) { assert(OCHY_Voter_State == 0);
}
if ((OCHY_comparator_state_ICH1 == 0) && (OCHY_comparator_state_ICH2 == 1)
 && (OCHY_comparator_state_ICH3 == 0)) { assert(OCHY_Voter_State == 0);
}
if ((OCHY_comparator_state_ICH1 == 0) && (OCHY_comparator_state_ICH2 == 0)
 && (OCHY_comparator_state_ICH3 == 1)) { assert(OCHY_Voter_State == 0);
}
if ((OCHY_comparator_state_ICH1 == 0) && (OCHY_comparator_state_ICH2 == 0)
 && (OCHY_comparator_state_ICH3 == 0)) { assert(OCHY_Voter_State == 0);
}}
```

Figure 1 The C code and the eight assertions verified with CBMC.

in VHDL did not directly correspond to the original pseudo code, but had `&&` (and) in the innermost level and `||` (or) in the outermost level.

The code shown is generated by the translator tool from the original VHDL source. In practice, with the active low reset signal, it would make more sense to use a falling edge instead of rising edge to trigger the code block. However, the block gets triggered with the next rising edge of the clock signal in any case, so this did not affect the verification task.

The code shown in Figure 3 shows a part that was added to the Verilog code only for the purposes of verification. There are now three new registers: `voter_state_check_in_pos`, `voter_state_check_in_neg`, and `voter_state_ check`. The value one of the first register should imply a voting result one. Correspondingly, the value one of the second register should

```
always @(posedge clk or posedge rst_n) begin
  if(rst_n == 1'b 0) begin
    voter_state_i <= 1'b 0;
  end else begin
    if((ICH1_comparator_state_och_in == 1'b 1 &&
        ICH2_comparator_state_och_in == 1'b 1) ||
       (ICH1_comparator_state_och_in == 1'b 1 &&
        ICH3_comparator_state_och_in == 1'b 1) ||
       (ICH2_comparator_state_och_in == 1'b 1 &&
        ICH3_comparator_state_och_in == 1'b 1))
    begin
      voter_state_i <= 1'b 1;
    end
    else begin
      voter_state_i <= 1'b 0;
    end
  end
end
```

Figure 2 The implementation of the voting code after VHDL to Verilog translation.

```
reg  voter_state_check_in_pos;
reg  voter_state_check_in_neg;
reg  voter_state_check;

initial begin
  voter_state_check_in_pos = 0;
  voter_state_check_in_neg = 0;
  voter_state_check = 1;
end
```

Figure 3 The added verification code in Verilog – part 1.

imply a voting result zero. The value of the third register should always be one if the system is working correctly. Since registers in Verilog have unknown initial values by default, the new registers are assigned initial values in the `initial` block.

Figure 4 shows the actual assertion block that gets triggered similarly to the original voting block. If the reset is not active and at least two of the inputs are one, the first new register gets value one. Correspondingly, if the reset is not active and at least two of the inputs are zero, the second new register gets value one. The third new variable gets assigned a value indicating whether the value of the first new register implies the voting result and the value of the second one implies the negation of the voting result.

One should note that the assignments are non-blocking, i.e. the right-hand side of each of the assignments is evaluated first. The assignment to the left-hand side is delayed until all the evaluations have been done.

The structure of the code block enables adding and removing "test cases" (input combinations in the context of the corresponding expected output value) from the statements and the expression `1'b 0` ensures that the assertions work also without any "test cases". We

```
always @(posedge clk or posedge rst_n)  begin
  voter_state_check_in_pos <= rst_n & (1'b 0
    | (ICH1_comparator_state_och_in & ICH2_comparator_state_och_in)
    | (ICH1_comparator_state_och_in & ICH3_comparator_state_och_in)
    | (ICH2_comparator_state_och_in & ICH3_comparator_state_och_in)
    );
  voter_state_check_in_neg <= rst_n & (1'b 0
    | (!ICH1_comparator_state_och_in & !ICH2_comparator_state_och_in)
    | (!ICH1_comparator_state_och_in & !ICH3_comparator_state_och_in)
    | (!ICH2_comparator_state_och_in & !ICH3_comparator_state_och_in)
    );
  voter_state_check <= (!voter_state_check_in_pos | voter_state_i) &
                       (!voter_state_check_in_neg | !voter_state_i);
  assert (voter_state_check);
end
```

Figure 4 The added verification code in Verilog – part 2 (please note the use of bitwise operators).

think that this is a robust, easy-to-use and reusable solution, since it allows extending the assertions with new properties incrementally. However, since the code in this case study is simple, the benefits are not so visible here.

The solution can be extended into more complex systems. For each bit of output two new registers and assignments to them are added, as well as corresponding terms to the expression of the final assignment. For each bit of input a new term is added to the relevant bitwise conjunctions of the assigned expressions for each expected output value. The assignment for an expected output value is placed into an always block corresponding to the situations where the value of the output may change in the code to be tested. However, this method is limited to stateless systems; a system with internal memory cannot be handled in such a straightforward manner.

One practical problem related to the inexperience of the person using the bounded model checking tools was that it was seemingly easy to verify properties that did not correspond to the actual requirement. For this reason we used a fault seeding technique where we introduced errors to the properties and checked whether it was possible to verify the erroneous properties. If not, we also checked that the counterexample provided by the tool corresponded to the seeded error. In practice, the errors seeded were more or less random changes made to the properties, i.e. we did not follow any systematic pattern. We think that this is a useful and practical technique for engineers without much experience in using model checking tools since it can be used to help determine whether a specification actually captures the desired intention, as is done with tests of vacuity [3, 2], where it is determined whether a subproperty is actually needed in the specification. This allows, for example, showing that an implication is true "by default" because the left side is always false.

All the assertions shown in the figures were verified with the tools. The bound value we used with EBMC was relative low, though. Once we became familiar with the tool, we noticed that the bound given to the tool as a command line option made a big practical difference. First, in some cases, it was possible to find problems in the assertions only when the value of the bound was high enough. This should be taken into account when using the fault seeding technique. Second, while the execution time of the tool with low bound values was reasonable (bound value 1000 corresponded roughly to 10 seconds in verification time

with a regular laptop computer), the execution took much more time with higher bound values due to the state space explosion. We also ran into some warning messaging concerning solver inconsistencies and one segmentation fault. Nevertheless, the tools were considered a good choice for the purposes of this small case study. However, especially the EBMC tool would be much more appealing from the practical point of view if a proper user manual and documentation were available.

Regarding future work, creating the test code as used in the case study can be cumbersome if inputs and outputs are numerous. More complicated inputs and outputs such as integers have to be handled bit by bit, which causes even more work. However, since the test code is very regular, it could be generated automatically with a suitable assisting tool. The registers and assignments can be created based on a list of outputs, with those of more complex types converted to a number of single-bit outputs. The expressions for the expected value assignments can be similarly created based on a listing of input combinations with the corresponding outputs, which may be given for example as a CSV (Comma Separated Values) file. In this way test cases can be converted into assertions in the code with little effort using Excel sheets created by test engineers, for instance.

4 Discussion

Even though our case study was small-scale in terms of the code checked, it helped us to identify some potential problems and partial solutions in the context of using model checking techniques to verify safety-critical systems. In more detail, the analysis of the results of the case study led to the following recommendations.

First, formal verification is seen useful at least in simple cases like the one studied. It was possible to develop a generic assertion mechanism for the code translated from VHDL to Verilog, which should be reusable in the verification of similar designs and further supported by assisting tools. Training would still be needed, though, in order to get engineers to use the tools.

While reusing assertions is seen to be beneficial, understanding how to develop effective assertions would need further training in the next step after basic training, unless this is solved by assisting tools. We think that starting with simple "test cases" before moving towards verifying more complicated properties can help in this process. In addition, we recommend using the fault seeding technique where errors are introduced to the properties for the purposes of checking whether it is possible to verify the erroneous properties; in our case this helped us to catch errors in the assertions.

Second, the tools used in this study worked well, but their scalability is still unknown. It would also be better if the VHDL code could be checked directly without the translation process to Verilog, unless a (certified) translator that could be trusted is found.

Third, the design flow in this particular case could be improved by specifying the properties associated with the requirements more precisely. This would allow detecting errors and inconsistencies already in the requirements capturing phase, as this phase is widely recognized to be critical. In an ideal case, the same properties could be translated into assertions used in the formal verification of the VHDL code. These kinds of properties and assertions could be reused in the case of modifications in a regression testing fashion; they could ease the burden of reverification needed in case of modifications that affect many elements. In addition, there might be some generic high level properties and assertions that could be used by different projects as sanity checks for a set of implementations sharing commonalities.

Fourth, experimenting first with tiny systems is highly recommended. Model checking

suffers from the state explosion problem like any other formal verification technique and with bigger systems more expertise is required to specify the system and requirements in a way that can be handled with the computing resources available. Moreover, complex specifications are more error prone to write and harder to check.

One practical problem related to the tools might be to find a suitable formal verification tool. Formal verification tools capable of analyzing VHDL exist, such as [13, 5]. Due to high license costs, however, it might be more economical to buy formal verification as a service (see, for instance [21]), if a company has only a limited need for such a tool. This option would also require less training. Another tool-related issue is certification: in principle, the software tools used in developing safety-critical systems should be certified by independent bodies [14, p. 83]. While certification is commonly used for compilers, we are not aware of any certified formal verification tool; this might become an issue in the future on high SIL/SC levels.

To conclude, the practical case study as well as the review of the related work show that model checking is a useful technique in the development of safety-critical systems. While there still are many problems to be solved, the tools are getting more scalable and user-friendly. In particular, it would be essential to provide tools that can work directly on the pseudo or source code used in the development and that require only basic training to be useful. Moreover, the whole development process could be streamlined with the support of such tools. While the standards regulating the development practices in the safety-critical domain are recommending the use of formal verification tools, the biggest problem seems to be related to training, and methodological introduction into the development process that could be eased with the help of simple assisting tools that, for instance, use input formats familiar to the users.

Acknowledgements

Funding from Tekes and the companies participating in the Ohjelmaturva project is gratefully acknowledged. The authors would also like to thank the anonymous reviewers for their helpful comments.

References

1 Angeletti, D., Giunchiglia, E., Narizzano, M., Puddu, A., Sabina, S.: Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. J. Autom. Reason. 45, 397–414 (December 2010), http://dx.doi.org/10.1007/s10817-010-9172-3

2 Ball, T., Kupferman, O.: Vacuity in testing. In: Beckert, B., Hähnle, R. (eds.) Tests and Proofs, pp. 4–17. LNCS 4966, Springer Berlin / Heidelberg (2008)

3 Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in ACTL formulas. Formal Methods in System Design 18, 141–162 (2001)

4 Björkman, K., Frits, J., Valkonen, J., Lahtinen, J., Heljanko, K., Hämäläinen, J.J.: Verification of safety logic designs by model checking. In: Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies, NPIC&HMIT (2009)

5 Cadence: Incisive Formal Verifier datasheet. http://www.cadence.com/rl/Resources/datasheets/IncisiveFV_ds.pdf, cited March 2011

6 CBMC, EBMC: Homepage. http://www.cprover.org, cited March 2011

7 Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new Symbolic Model Verifier. In: CAV'99. pp. 495–499. LNCS 1633, Springer (1999), http://nusmv.itc.it

8 Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA (1999)
9 Dammag, H., Nissanke, N.: Safecharts for specifying and designing safety critical systems. In: In Symposium on Reliable Distributed Systems. pp. 78–87 (1999)
10 Gonzales, M.: VHDL to Verilog RTL translator v2.0. http://www.ocean-logic.com/downloads.htm, cited March 2011
11 Hatcliff, J., Dwyer, M.: Using the Bandera Tool Set to model-check properties of concurrent Java software. In: Larsen, K.G., Nielsen, M. (eds.) Proc. 12th Int. Conf. on Concurrency Theory, CONCUR'01. pp. 39–58. LNCS 2154, Springer-Verlag (2001)
12 Hsiung, P.A., Chen, Y.R., Lin, Y.H.: Model checking safety-critical systems using safecharts. IEEE Transactions on Computers 56, 692–705 (2007)
13 IBM: IBM RuleBase homepage. http://www.research.ibm.com/haifa/projects/verification/RB_Homepage/, cited March 2011
14 International Electrotechnical Commission: IEC 61508-7, Functional safety of electrical/electronic/programmable electronic safety-related systems, part 7 (2010), ed2.0
15 International Electrotechnical Commission: IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems (2010), ed2.0
16 Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodova, A., Taylor, C., Frolov, V., Reeber, E., Naik, A.: Replacing testing with formal verification in Intel® Core™ i7 processor execution engine validation. In: CAV 2009. LNCS 5643, Springer (2009)
17 Kaivola, R., Kohatsu, K.R.: Proof engineering in the large: Formal verification of Pentium 4 floating-point divider. In: Margaria, T., Melham, T.F. (eds.) CHARME. pp. 196–211. LNCS 2144, Springer (2001)
18 Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag (1991)
19 Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. of the 38th Design Automation Conference, DAC'01. pp. 530–535 (2001)
20 de Moura, L., Bjorner, N.: Z3: An efficient SMT solver. In: Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340. LNCS 4963 (2008)
21 NoBug: Homepage. http://nobug.ro, cited March 2011
22 Pentium FDIV bug. http://en.wikipedia.org/wiki/Pentium_FDIV_bug
23 SATabs: Homepage. http://www.cprover.org/satabs/, cited July 2011
24 Storey, N.: Safety Critical Computer Systems. Addison-Wesley (1996)