Automated Formal Verification and Testing of C Programs for Embedded Systems *

Susanne Kandl, Raimund Kirner, Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A-1040 Wien, Austria
{susanne, raimund, peter}@vmars.tuwien.ac.at

Abstract

In this paper we introduce an approach for automated verification and testing of ANSI C programs for embedded systems. We automatically extract an automaton model from the C code of the SUT (system under test). This automaton model is on the one hand used for formal verification of the requirements defined in the system specification, on the other hand, we can derive test cases from this model, for both methods we use a model checker. We describe our techniques for test case generation, based on producing counterexamples with a model checker by formulating trap properties. The resulting test cases can then be applied to the SUT on different test levels. An important issue for model checking C-source code, is the correct modeling of the semantics of a C program for an embedded system. We focus on challenges and possible restrictions that appear, when model checking is used for the verification of C-source code. We specifically show how to deal with arithmetic expressions in the model checker NuSMV and how to preserve the numerical results in case of modeling the platform-specific semantics of C.

1 Introduction

Due to the increasing capacity of processors used in embedded systems, the complexity of the applied software is growing. Thus, more and more effort is needed for testing embedded systems software. Beside conventional testing techniques [6], new testing methods are starting to gain ground. The goal is to optimize the effort of testing and thus it is an aim to *automate* most parts of the testing process. According to safety-relevant embedded systems applications, safety-standards like the IEC 61508 [25] apply that define strict conditions for the software quality and the demanded testing process. One way for automated verification and test case generation is based on formal methods. The SUT (system under test) is described as an automaton model. A model checker is used to verify the properties defined in the system specification. In our approach we automatically extract the model from the C code of the SUT. Verifying a property on the model means, that this property holds on the C-source code. Within the model extraction process we have to deal, amongst other issues, with transforming arithmetic operations, appearing in the application, correctly to the automaton model. In a second step we use the model for deriving test cases automatically, these test cases can then be applied to the SUT. Depending on the execution environment of the produced test cases, we can find failures of the different representations of the SUT.

The article is organized as follows: In Section 2 we describe our 2-step approach for verification and testing of the system. In the Section 3 the model extraction process is demonstrated on an example. Section 4 refers to the formal verification step. In Section 5 we explain our techniques for test case generation. Section 6 deals with the challenges when a model checker is used for the verification of C programs. In Section 7 we discuss the solution, how we deal with arithmetic operations in the model checker NuSMV. Subsequently we present our preliminary results and give an overview on related work. Finally we conclude with a summary.

*This work has been partially supported by the FIT-IT research project "Systematic test case generation for safety-critical distributed embedded real time systems with different SIL levels (TeDES)"; the project is carried out in cooperation with TU-Graz, Magna Steyr, and TTTech.

2 Verification Process

2.1 2-Step Approach

The verification of the SUT is realized in two important independent steps:

In the first step the platform-independent semantics of the system can be verified formally by model checking. The automaton model extracted from the source code can be used to verify the properties from the system specification, because the automaton model is just another representation of the platform-independent semantics of the C program. By verifying all requirements from the specification, we can show that the C program conforms to the specification. If a requirement is not valid on the model, the implementation has to be corrected. This step in the verification process proves if the program behavior conforms to the specification.

The second step is testing the system by execution of test cases on the target platform. For this purpose, we are generating test cases by means of model checking from the model of the system. The execution of the test cases on the target platform proves, whether the platform-specific semantics of the program has the same behavior as the model. We proved the correctness of the model in the first verification step. Showing that the system, running on the target platform, conforms to the model behavior proves that the platform-specific behavior conforms to the specification.

2.2 Automatic Verdict and Test Case Application

To realize a testing procedure with automatic *verdict*, one needs test cases, i.e., a structure, consisting of input data and expected output. The big advantage of our framework is, that the expected output is also determined automatically.

The C-source code for the SUT is generated by a code generator from the Matlab Simulink model of the SUT. For the generation of test cases we also use the automaton model that is automatically extracted from the C-source code. That means that our model maps directly to the C-source code, resp. to the Matlab Simulink model (assuming that no errors occur during the code generation process). When using the same model for test case generation and code generation, the verdict is deduced from a model that maps *completely* to the code. In the first phase of our verification and testing approach we show by formal verification that the automaton model, that is just another representation of the C-source code (generated from the Matlab Simulink model) is correct in respect to the specification. This means, we show by formal verification that the Matlab Simulink model and the corresponding C-source code is correct. In the second phase we derive test cases from the automaton model. Because we have shown the correctness of the extracted model, we know that the verdict given by the produced test cases is also correct. The resulting test cases can be applied to the Matlab Simulink model (testing on *model in the loop (MIL)*-level), can be run against the object code of the SUT (testing on *software-in-the-loop (SIL)*-level), the test cases can also be run on the processor (testing on *processor-in-the-loop (PIL)*-level) or within the hardware environment (testing on *hardware-in-the-loop (HIL)*-level).

All the test cases executed on the Matlab Simulink model should pass the test procedure, because all the behavior from the model was transformed within the code generation process to the C-source code and then represented in the automaton model, from which we derived the test cases with the verdicts. Problems with test cases, that fail the test procedure on the Matlab Simulink model, may stem from the simulation environment. For instance, the automaton model is derived from the C-source code generated by TargetLink from dSPACE[1] and the Matlab Simulink simulation is based on the Real-Time Workshop[2]. Running the test cases on the object code should also result in no failures. But also an erroneous behavior of the object code may be identified caused by the compiler, typically due to optimization mechanisms. The simulation of the Simulink model is, in general, not realized using the same compiler as for generating the object code for the target platform. If there are differences in the system behavior depending on the used compiler, this is a strong evidence that the compilation process influenced the system behavior. Running the test cases on the target platform proves, whether the actual behavior differs from the behavior, predicted by the model. This can, for instance, be caused by deviations due to the target-execution on the processor. For instance, a 32-bit floating-point calculation in the software is reduced to 16-bit-fixed-point values and this influences the system behavior on the target platform significantly. At the last stage, testing is realized within the hardware environment. At this stage dependencies within the hardware environment (for instance caused by the bus system) can cause that test cases fail.

Figure 1 shows the test case application and the factors, that can cause failures on the different testing levels.

[1] `http://www.dspace.com`
[2] `http://www.mathworks.com`

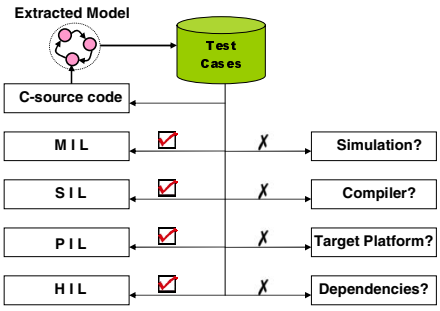

Figure 1. Test Case Application

3 Model Extraction

For the 2-step verification process described above we need an automaton model for the system under test. This automaton model is built from the C-source code. The model is just another representation of the behavior of the C program.

3.1 Basic Principles

The model extraction is done in the following steps:
1. First the C-source code is parsed and by static analysis the *syntax tree* of the program is generated.
2. The syntax tree is used to generate the *automaton model* of the system. This is realized by sequentially processing the syntax tree and interpreting the semantics of the basic statements of the syntax tree.
3. The description of the automaton model is given in an *automata language* (for instance, the modeling language of the NuSMV[3] model checker).

The described model extraction technique is, basically, straight forward. However, as described in Section 7, special care must be taken to ensure correct modeling of arithmetic expressions. At the moment we are able to extract the model for a subset of standard ANSI C. Currently, we do not support extraction of control structures like loops or function calls. Function calls can be handled by in-lining.

3.2 Example

As an example, consider the small C program shown in Listing 1. In this program the function *test* takes two arguments x and a and contains two composed if-else conditions. Depending on the current values of x and a, the variable x is assigned with the values 10, 20, or is incremented. The resulting syntax tree consists of approximately 100 nodes, of which we have to deal with 16 basic statements for the generation of the automaton model. The analysis and interpretation of the syntax tree yields the automatically generated NuSMV model, see Listing 2: First the variables are defined (lines 3 to 5). The additional variable `sequence_nr` in the NuSMV model stems from the static analysis and represents the program counter. Starting in the initial state `sequence_nr = 16`, there are two transition blocks, one for the `sequence_nr` and one for the variable `v0_x`. In the lines 14 and 15 the first if-query of the source code is checked. Depending on the assignment of `v0_x`, the `sequence_nr` changes to 2 and the value of `v0_x` is assigned with 10 (line 18 of the Listing 2) or the next query (`else if`) is checked (lines 12 and 13 of the Listing 2). In this way the execution of the C program is described as a transitional system. This NuSMV model can be directly processed by the NuSMV model checker. In the following sections we describe how to verify this model and generate test cases from it.

```
1 int test(int x, int a) {
2   if (x == 1) {
3     x=10;
4   } else if (a == 2) {
5     x=20;
6   } else {
7     x=x+1;
8   }
9 }
```

Listing 1. C source code

4 Formal Verification

For the formal verification of the system the properties from the specification have to be translated into temporal logic formulas (e.g., computation tree logic *CTL*). These formulas can be verified on the model with a model checker.

Some properties from the specification are suitable to be checked directly on the extracted model. (e.g., reachability properties). In contrast to, other functional properties have to be transformed to be suitable for the verification process. If we consider a reactive system, the function presented by the extracted model is called in a loop. A function is terminated by setting the `sequence_nr` to 1. Thus the reactive loop can be closed by resetting the counter `sequence_nr` at the exit points of the function not to 1, but to the value

[3] `http://nusmv.irst.itc.it/`

```
MODULE main
VAR
  sequence_nr: 0..255;
  v0_x: 0..255;
  v1_a: 0..255;
ASSIGN
  init(sequence_nr):= 16;
  next(sequence_nr):= case
    sequence_nr= 2: 1;
    sequence_nr= 5: 1;
    sequence_nr= 8: 1;
    sequence_nr= 12 & (v1_a=2) : 5;
    sequence_nr= 12 & !(v1_a=2) : 8;
    sequence_nr= 16 & (v0_x=1) : 2;
    sequence_nr= 16 & !(v0_x=1) : 12;
  esac;
  next(v0_x):= case
      sequence_nr= 2: 10;
      sequence_nr= 5: 20;
      sequence_nr= 8: v0_x + 1;
  esac;
```

Listing 2. NuSMV model

of the `sequence_nr` at the init-state, in our example to `sequence_nr = 16`. The step of formal verification of the properties from the specification on the model proves, whether the properties are valid on the automaton model of the SUT. Because the model is directly extracted from the ANSI C source code, we show in the formal verification step that the C-source code is correctly implemented referring to the specification. In this verification step we can not validate any behavior of the object code, or any behavior of the object code executed on the target platform. Thus, to verify the correct execution on the target platform, we need test cases for testing purposes. In the next section the process of test case generation is described. These test cases can be applied to the object code running on the target platform.

5 Test Case Generation

For the test case generation we also use model checking techniques. Beside its original purpose of formal verification of systems, model checking has become an applicable tool for test case generation. The main purpose of a model checker is to verify a formal property on a system model. In case that the formal property is invalid on a given model, a model checker provides a *counterexample*, which describes a concrete path on which the property is violated. This feature of a model checker can be used to generate test cases in a formal and systematic way.

For finding suitable test cases the challenge is to find appropriate properties, that yield specific paths we can use as test cases. This is realized by the formulation of so-called *trap properties*. Trap properties are inconsistent with the model and force the model checker to produce counterexamples. That means, a property f formally states that a certain location in the model cannot be reached. The counterexample generated by the model checker provides a trace to reach this location.

First, we have to identify the variables we are interested in to test. For example, we are interested in assignments to the variable x because a requirement from the specification tells us, that this is a critical item to test. The interesting instances of x are calculated by a prior static program analysis phase. Based on the formulation of the property f the model checker can find those paths that will reach the location of x. This approach provides a systematic way to find test cases for testing the assignments to x.

As an example, lets assume we are interesting in testing the assignment $x = 20$ in our program in the Listing 1 in Section 3.2. From our model extraction we know that this assignment corresponds to the state `sequence_nr = 5` in the automaton model. So we can formulate the following property:

```
PSLSPEC G(sequence_nr=5)->!(F sequence_nr=1)
```

Providing this property to the model checker results in a counterexample that represents a trace to test the assignment of the value 20 to the variable x. The resulting test cases can be applied to the system under test.

6 Challenges in Model Checking C Code

Applying model checking techniques to C code is, basically, no novel method. There are lots of works describing the application of model checking for verifying and testing C programs, for instance Henzinger et al. [23], Clarke and Kroening [10], or Chen et al. [9]. Also our testing framework has been successfully applied to a case study from the automotive domain supplied by one of our industry partners. But within our project, we were confronted with lots of C-specific attributes, that may be a problem for model checking. Only few works are concerned with the topic, which challenges appear, when using a model checker for verifying C-source code and what possible restrictions have to be considered. Schlich and Kowalewski [36] present an interesting comparison and evaluation of model checkers for embedded systems. According

to that work and reflecting the experiences within our project, we want to summarize the items that have to be kept in mind, when using model checkers for C-source code.

6.1 C-Specific Semantics

There are model checkers, that are able to process C-source code directly, for instance BLAST [23] or CMBC [12]. These model checkers support ANSI C or a subset of it. For most other model checkers the C-source code has to be transformed into a model described in the automaton language of the model checker, examples for this kind of model checkers are SAL [16] or SMV [29]. Both kinds of model checkers are not suitable to handle all constructs used in C programs for embedded systems.

- Hardware and Compiler Specific Behavior: As we have already mentioned in Section 2 the C-source code only describes the platform-independent behavior of the program. Although, we have to consider also platform-specific attributes when building the model (e.g., a variable of the data type *int* has another representation in a 16-bit architecture, than in a 32-bit architecture), all hardware- and compiler-specific attributes can not be formally verified on the model of the system based (only) on analysis of the C-source code. One attempt may be to integrate compiler specific behavior into the model. This works for influences caused by the compiler that are known and that can be easily modeled. Many adjustments (e.g. optimization) are done by the compiler without detailed knowledge of the resulting effects on the object code. In our testing framework we cover these compiler-specific semantics by the execution of the generated test cases.

- Embedded language statements: To enforce the performance of programs closed to hardware, many applications for embedded systems include parts of assembly language statements. These constructs are typically not supported by C-model checkers, that can process ANSI C.

- Timing Behavior: Depending on the specification of the system, timing constraints can be part of the functional requirements. Abstract time can be integrated into the model of the system by introduction of an additional timing variable. Constraints referring to some timing dependencies, for instance a variable is assigned *after* the execution of a function, can be formulated in CTL and verified on the extended model. Timing constraints according to real time are not suitable to be verified with a model checker like NuSMV. There exist special model checkers, based on timed automata, like Uppaal[4]. So far, as we experienced with Uppaal, only small models can be handled conveniently. Timing behavior has to be tested by further analysis, e.g. WCET (worst-case estimation time).

- Floating Point Arithmetic: Model checking code with floating-point arithmetics is challenging. This is not because of state explosion, since a 32-bit floating-point variable causes the same state space as a 32-bit integer variable. However, the problem is that modeling state transitions of floating-point variables is quite expensive: one has to model the semantics of the floating-point unit to explicitly update the fields sign, mantissa, and exponent of a floating-point variable. In contrast, modeling integer transitions is much more simple, as each modeling language already provides integer operators, which only have to be adjusted for the concrete value range of the data type. Thus, model checking integer operations is typically faster by a linear factor.

- Casting: In C programs casting is a common way to change the data type. In model checkers like NuSMV the data types are declared once in the header of the model. A variable that is casted in a C program as a *signed int* and then assigned to a variable of type *unsigned int* can cause problems in the model and results in an error messages for incompatible data types. A possible solution may be to introduce additional variables to save the variables of the new data types, but this leads to the effect, that far more variables have to be defined as variables existing in the C program. This causes additional state space explosion.

- Pointers: Model checking programs with arbitrary use of pointers is very challenging. In this case, one has to model each variable by its name and its memory location. The memory itself becomes a variable (large array) within the model. Since the bit-space of the whole data memory is typically much larger than the bit-space of all program variables together, this results in a significant increase of the state space. Thus, modeling arbitrary use of pointers is almost infeasible. However, pointers are often used in a restrictive way, such that this behavior can be modeled without the need of modeling the whole data memory. For example,

[4] `http://www.uppaal.com/`

the following use cases of pointers can be modeled in an abstract way:

Call-by-reference: when a function takes a pointer as an argument and this pointer is only used to read and access a variable in the scope of the caller, there is no need to model the whole state space of the memory.

Function pointer: when a function pointer is only directly assigned with function addresses and this pointer is only used to call the function, there is no need to model the state space of the memory. Instead, at each function call using the pointer one has to model the call by checking of equivalence to all possibly assigned functions within the program.

Pointer-access to arrays: in general, as long as a pointer is used to address elements within a certain object, one does not have to model the state space of the whole memory. For example, when a pointer is used to access elements of a data array, one can model the pointer by a symbolic base address (in this case the array name) and a numerical offset. To ensure that this abstraction is correct, one has to show that the offset of the pointer will never go beyond the array limits. This property can be directly checked using the model.

C-Model Checkers, like CMBC, support pointers in a limited way [11].

6.2 Model Checking Specific Restrictions

The main drawback of using model checkers for C software is the state space explosion for big data domains. OBDD (ordered binary decision diagram)-based model checkers are building the automaton for the whole data space. This yields in large models. A solution can be to use *bounded model checking*, where the state space is created and searched only to a specific depth of the underlying Kripke-structure defined by a *bound*. A natural restriction by using bounded model checking for the verification of systems is, that errors can be missed, if the bound is not chosen properly. Another way to deal with the state space explosion is using abstraction techniques. A common abstraction techniques is *predicate abstraction* [13]. Still this approach lacks on some restrictions [28]: limited number of supported C constructs or negligence of possible arithmetic overflows.

7 Modeling in NuSMV

Out of the multitude of available model checkers, we have chosen the model checker NuSMV for our project, mainly because of two reasons: NuSMV is open source, so adaptations to the source code of the model checker can be done if necessary and NuSMV supports an extension of CTL for the formulation of the properties as an IEEE standard, namely SUGAR[5]. NuSMV can not directly process C-source code, so the program has to be transformed into the automaton language of NuSMV. We described this step detailed in the section 3. In the following we describe, how we solved the problem of dealing with arithmetic expressions appearing in the C program within NuSMV.

7.1 Modeling Arithmetic Operations

Modeling the semantics of ANSI C operations in the automaton language SMV, used in the model checker NuSMV, has to be done carefully to preserve the numerical results of arithmetic operations. There are two aspects one has to be aware of:

- The semantics in the value domain of the ANSI C language is not completely defined by the standard. For example, the width of basic data types, the evaluation order of most operators, the memory layout, or memory alignment is not defined. Each compiler or interpreter has to take several implementation choices to complete the semantics in the value domain. Thus, when analyzing or modeling an ANSI C program, one has to be aware of the properties of the concrete execution platform. We therefore differ between the *platform-independent semantics* and the *platform-specific semantics* [27].
- When modeling a C program by a formal modeling language, it is the common case, that the data types and operators of ANSI C do not directly match those provided by the modeling language. As already mentioned above, the width of ANSI C data types depend on the concrete execution platform. The integer data type of SMV is implemented to be in the range of $2^{-31} \dots (2^{31} - 1)$. Though SMV allows to specify subranges of integer types, this is not sufficient to model the C data types. The problem is that specifying integer subranges in NuSMV does not imply that the operands also work with *modular arithmetic* in that range.

7.1.1 Out-Of-Boundary Problem of NuSMV

Besides the numerical correctness of arithmetic operations, there is also another issue why arithmetic operations require some adaptations when modeling them

[5] `http://www.haifa.ibm.com/projects/verification/sugar/index.html`

with NuSMV. Arithmetic operations in NuSMV can cause the *out-of-boundary-problem*. That is that the *arithmetic decision diagram* (ADD) is built for all possible values of a variable. For instance, incrementing the maximum value of a variable results in a value out of the data domain of this variable.

This behavior is caused by the way how NuSMV interprets an arithmetic expression: The ADD is constructed for the right-hand side of an expression, like a *binary decision diagram* (BDD) but the leaves are the possible values that could be assigned to the left hand side of an expression. If there is a leaf in this ADD that is out of range for the left hand side then an error is reported. In BDD-based model checking the (actually never-claimed) values will disappear when the whole model is built.

7.1.2 Possible Solutions

The following adaptation techniques of arithmetic operations are solutions to the above mentioned modeling problems:

Formulation of additional conditions:
: Additional queries are added to cover critical conditions, e.g., value < `var_max`. Such conditions can be easily formulated for simple arithmetic operations. However, in case of complex arithmetic expressions it can be hard to find appropriate queries.

Adapting the transitions to boundary semantics (*BOUND*):
: By the means of handling the critical values in separate transitions the minimum and maximum values of a variable are handled specially. Again in complex expressions identifying and trapping the critical cases can be difficult. Take into account, that regarding to the numerical correctness, the above introduced techniques are only applicable if the concrete arithmetic expression of the C code is free of overflow, i.e., the modular arithmetic of C is not used.

Integration of modulo operators (*MOD*):
: Integrating an additional modulo operator works for BDD-based model checking. The integration of a modulo operator includes edges that are never claimed, but they will disappear when the whole model is build. In the case of *bounded* model checking the extra clauses persist and causes not-existing paths. Furthermore, regarding the numerical correctness, this technique is only correct, if the arithmetic expression in the C code consists only of unsigned types, or it is free of overflow.

Adapting the transitions to modulo semantics (*MODULO*):
: Again the critical values are treated in separate transitions. If the value of a variable is out of the data range the value is set to the min/max-value of the variable. This semantic provides the same numerical results as the overflow handling in the ANSI C programming language. For details of the conversion of arithmetic operands in C refer to the ISO/IEC 9899 Standard for the Programming Language C [26], page 36 et seqq., resp. the semantics of expressions in [26], page 58 et seqq. Thus, all arithmetic expressions of ANSI C programs where overflow can occur, have to adapted with this technique.

7.1.3 Examples of Modeling Arithmetic Operations

In the following the algorithms to model arithmetic expressions of ANSI C are exemplary given for the binary operator '+' (plus). Variables are denoted with `varname`, the range of a variable is given by `var_min` to `var_max`.

- Algorithm for BOUND:
  Generic scheme:
  sequence_nr=s&expression<(`var_max`+ 1):expression
  sequence_nr=s &expression>`var_max`:`var_max`
  Line 20 of Listing 2 will be adapted to:
  ```
  sequence_nr=8&(v0_x+1)<(255+1):v0_x+1;
  sequence_nr=8&(v0_x+1)>(255+1):255;
  ```

- Algorithm for MOD:
  Generic scheme:
  sequence_nr=s:expression mod(`var_max`+1)
  Line 20 of Listing 2 will be adapted to:
  ```
  sequence_nr=8:v0_x+1 mod(255+1);
  ```

- Algorithm for MODULO:
  Generic scheme:
  sequence_nr=s&expression<(`var_max`+1):expression
  sequence_nr=s&expression>`var_max`:(expression-var_min) mod(var_max-var_min+1)+var_min
  Line 20 of Listing 2 will be adapted to:
  ```
  sequence_nr=8&(v0_x+1)<(255+1):v0_x+1;
  sequence_nr=8&(v0_x+1)>(255+1):(v0_x+1-0)
  mod(255-0+1)+0;
  ```

To summarize, the MODULO adaptation technique is required in the general case. The concrete values of `var_min` and `var_max` depend on the concrete execution platform, i.e., the implemented semantics of the compiler or interpreter.

8 Preliminary Results and Outlook

So far we have successfully implemented a prototype for the model extraction with the integrated handling of the modulo semantics for arithmetic operations. The model extraction works for a subset of C (for instance, function calls are only supported by in-lining the functions). We can generate test cases by manual formulation of the introduced trap properties. The test framework has been successfully applied to a case study from the automotive domain. Our future work will concentrate on the automated formulation of the necessary properties to find test cases that map to a specific functional property from the system specification. Once we have identified rules and algorithms for the automatic formulation of the trap properties, we will implement these algorithms and integrate them into our test case generation framework.

9 Related Work

For on overview on general testing techniques see Harold: Testing - A Roadmap [22]. Testing techniques especially for embedded systems are described detail in the book from Broekman and Notenboom [6]. Referring to formal test methods Tretmans gives a introductory overview in [38]. The thesis of Mirko [15] discusses testing embedded systems on examples especially from the automotive domain. Our considerations about the automatic verdict of the test cases derived from the model are motivated by methodological issues on model-based testing, described by Pretschner et al. [33]. Our work on model extraction is based on works from Wenzel et al. [39]. Similar ideas can be found in Holzmann et al. [24]. Schroder et al. [37] discusses a few interesting modeling aspects for the automated test case generation. Beside our approach of extracting the automaton model from C-source code, there are some model checkers that can directly process C code, for instance BLAST (described in [23]) or CMBC [12]. For a general introduction to model checking techniques please refer to the book of Clarke et al. [14]. Model checkers can be also used for software testing [1, 7]. Model checking especially for embedded systems is discussed by Brinksma [5]. The idea of using model checkers for the generation of test cases aims back to the mid's of the '90, amongst others from Callahan et al. [7]. Ammann et al. [2] and Black [4], resp. Gargantini and Heitmayer [20] published works dealing with using model checking to generate test cases from system specifications. Coverage-based test case generation methods using model checkers are described from Rayadurgam et al. [35]. Recent works concerning automated test case generation with model checkers are from Okun et al. [30, 31], resp. Hamon et al. [21] using the model checker SAL. In principle all these works concern to the formulation of trap properties to specify the paths the model checker should deliver as a counterexample. Engels el al. [17] refer to this idea with the term *never-claims*. Automated testing focused on the automotive perspective is also described by Ranville and Black in [34]. Case studies and evaluation of the model-based test case generation techniques are from Chandra et al. [8], Pretschner et al. [3, 18] or Paradkar [32]. Corresponding to the manifold approaches to automate the testing procedure, a lot of tools for testing are provided. M.Broy et al.(Eds.) give an overview on available tools [19]. So far, as we know, the above mentioned literature does not discuss how to precisely model arithmetics of ANSI C operators with a formal modeling language.

10 Summary and Conclusion

We described a verification approach for automated verification of ANSI C programs. This approach uses a formal model for both, formal verification and test case generation. The model itself is extracted from the C code. We discussed the issues that may be critical when using model checkers for the verification of C programs for embedded systems. Using a formal modeling language as input to the model checker, it is important that the arithmetic computations of the C code are precisely modeled. We described special adaptation rules to precisely model the *modular semantics* of ANSI C. Due to the incomplete specification of ANSI C, this adaptation rules are platform-specific. Further, we have shown how the obtained formal model can be used to formally verify the program. We described also how to generate test cases automatically in a systematic way to complement the formal verification by testing.

References

[1] P. Ammann and P. E. Black. Model checkers in software testing. Online: http://citeseer.ist.psu.edu/ammann02model.html.

[2] P. Ammann, P. E. Black, and W. Majurski. Using Model Checking to Generate Tests from Specifications. In *ICFEM*, 1998.

[3] A.Pretschner, O.Slotosch, H.Lötzbeyer, E.Aiglstorfer, and S.Kriebel. Model based testing for real: The inhouse card case study. In *Proc. of FMICS 2006*, pages 79–94, 7 2001.

[4] P. E. Black. Modeling and Marshaling: Making Tests From Model Checker Counterexamples. In *Proc. of the 19th DASC)*, volume 1, pages 1B3/1–1B3/6, 2000.

[5] E. Brinksma and A. Mader. Model checking embedded systems desings. In *Proc. of the Sixth International Workshop on Discrete Event Systems*, Oct. 2003.

[6] B. Broekman and E. Notenboom. *Testing Embedded Software*. Addison-Weseley, 2003.

[7] J. Callahan, F. Schneider, and S. Easterbrook. Automated Software Testing Using Model-Checking. In *Proc. 1996 SPIN Workshop*, August 1996. Also WVU Technical Report NASA-IVV-96-022.

[8] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *ICSE '02*, pages 431–441, New York, NY, USA, 2002. ACM Press.

[9] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code, 2004.

[10] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.

[11] E. Clarke and D. Kroening. ANSI-C bounded model checker user manual. Technical report, School of Computer Science, Carnegie Mellon University, 2006.

[12] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[13] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, September–November 2004.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[15] M. Conrad. *Modell-basierter Test eingebetteter Software im Automobil: Auswahl und Beschreibung von Testszenarien*. DUV, 2004.

[16] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *LNCS*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[17] A. Engels, L. M. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *Proc. 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in LNCS, pages 384–398. Springer, 1997.

[18] P. et al. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*. IEEE, 2003.

[19] M. et al.(Eds.), editor. *Model-Based Testing of Reactive Systems, LNCS 3472*, Chapter 14 Tools for Test Case Generation, pages 391–438. Springer-Verlag Berlin Heidelberg, 2005.

[20] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests From Requirements Specifications. In *7th European Software Engineering Conference*, pages 146–162, 1999.

[21] G. Hamon, L. de Moura, and J. Rushby. Generating Efficient Test Sets with a Model Checker. In *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, 2004. Submitted for publication.

[22] M. J. Harrold. Testing: A roadmap. In *In Future of Software Engineering, 22nd ICSE*, June 2000.

[23] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast, 2003.

[24] G. J. Holzmann and M. H. Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification & Reliability*, 11(2):65–79, 2001.

[25] IEC. IEC 61508 functional safety of electrical/-electronic/programmable electronic safety-related systems. Technical report, IEC, 1998.

[26] International Standards Organisation. *ISO/IEC 9899:1999 Programming Languages - C*. American National Standards Institute, New York, 2nd edition, Dec. 1999. Technical Committe: JTC 1/SC 22/WG 14.

[27] S. Kandl, R. Kirner, and G. Fraser. Verification of platform-independent and platform-specific semantics of dependable embedded systems. In *Proc. 3rd International Workshop on Dependable Embedded Systems*, pages 21–25, Leeds, UK, October 2006.

[28] D. Kroening and E. Clarke. Checking consistency of C and Verilog using predicate abstraction and induction. In *Proceedings of ICCAD*, pages 66–72. IEEE, November 2004.

[29] K. McMillan. The SMV lanugage. Technical report, Cadence Berkeley Labs, March 23 1999.

[30] V. Okun, P. Black, and Y. Yesha. Testing with model checkers: Insuring fault visibility, 2003.

[31] V. Okun and P. E. Black. Issues in software testing with model checkers, 2003.

[32] A. Paradkar. Case studies on fault detection effectiveness of model based test generation techniques. In *A-MOST'05*. ACM, 2005.

[33] A. Pretschner and J. Phillips. *Model-Based Testing of Reactive Systems, LNCS 3472 (Eds.: M.Broy et al.)*, Chapter 10 Methodological Issues in Model-Based Testing, pages 281–291. Springer-Verlag Berlin Heidelberg, 2005.

[34] S. Ranville and P. E. Black. Automated testing requirements - automotive perspective. In *The Second International Workshop on Automated Program Analysis, Testing and Verification*, May 2001.

[35] S. Rayadurgam and M. P. Heimdahl. Coverage Based Test-Case Generation Using Model Checkers. In *Proc. of the 8th Annual IEEE Int. Conference and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, 2001.

[36] B. Schlich and S. Kowalewski. Model checking C source code for embedded systems. In *Proceedings of ISoLA 2005*. NASA/CP-2005-212788, 2005.

[37] P. J. Schroder, E. Kim, J. Arshem, and P. Bolaki. Combining behavior and data modeling in automated test case generation. In *Proceedings of QSIC'03*. ACM, 2005.

[38] J. Tretmans. Testing techniques, 2002. http://www.cs.auc.dk/~kgl/TOV04/tretmans-notes.pdf.

[39] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *DATE '05*, pages 606–611. IEEE, 2005.