Verification of Embedded Systems Using Modeling and Implementation Languages

Mathias Soeken[1,3], Heinz Riener[2],
Robert Wille[1], Görschwin Fey[2], and Rolf Drechsler[1,3]

[1] Institute of Computer Science, University of Bremen
Group of Computer Architecture, D-28359 Bremen, Germany
[2] Institute of Computer Science, University of Bremen
Group of Reliable Embedded Systems, D-28359 Bremen, Germany
[3] Cyber-Physical Systems
DFKI GmbH, D-28359 Bremen, Germany
{msoeken,hriener,rwille,fey,drechsle}@informatik.uni-bremen.de

Abstract. We propose a verification flow which aims at functional verification of embedded systems. We assume that the system is formally specified in a modeling language and implemented in a high-level programming language. The verification flow proceeds in two stages: first, we check for behavioral properties which indicate bad states at the level of the specification. Second, we prove that the individual components of the system conform to their specification. The first stage allows for the detection of design flaws in the system without considering implementation details, whereas the second stage refines the verification task allowing for the detection of functional bugs in the implementation. In both stages, the description of the verification task is automatically generated from the modeling language and the implementation language, respectively.

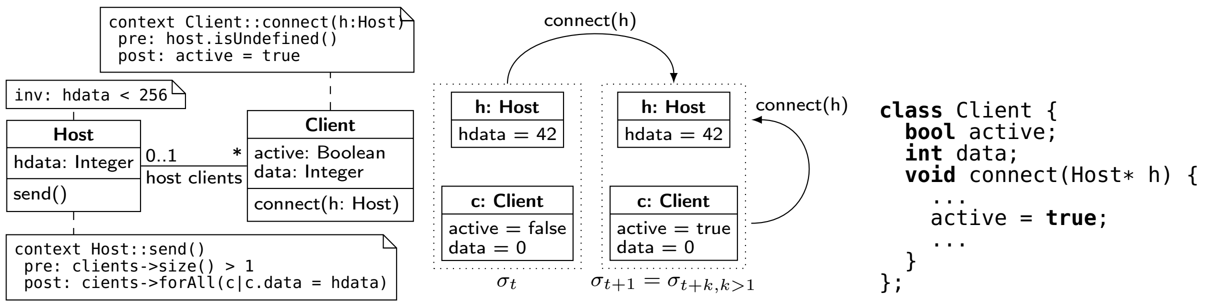
1 Introduction

Guaranteeing functional correctness of embedded systems is time-consuming and challenging. In modern system design flows, checking for functional correctness, i.e., *functional verification*, accounts for more than half of the total development time [1]. Due to time-to-market constraints, however, push-button approaches are desirable.

Formal methods aim at automatically checking whether a system implementation conforms to a formal specification. One such method is *model checking* [2, 3] which exhaustively explores the state space of a system implementation and generates a counterexample if and only if the formal specification does not hold. Model checking has successfully been used to prove functional correctness of hardware designs at the gate and register-transfer levels. Today, research focuses on lifting functional verification to higher abstraction levels, e.g., the *Electronic System Level* (ESL) [4–6]. At the ESL, a designer is not only faced with guaranteeing functional correctness of an isolated component but has to consider multiple components interacting with each other and their environment. The main challenge at the ESL is to tackle the possibly large state space to overcome the *state-explosion problem*.

Previous work includes *assume-guarantee reasoning* [7] and *synthesis from specification* [8, 9]. Assume-guarantee reasoning attempts to prove or refute a property by utilizing the specifications of individual components to allow for a step-wise refinement of the property. Synthesis generates a system which is correct-by-construction from a formal specification. Both techniques, however, are computational expensive. Moreover, providing a formal specification which entirely describes the functionality of a system is challenging for larger system designs.

In this paper, we propose a functional verification flow focusing at the ESL by lifting parts of the verification technique to a meta-modeling level. Given a formal specification



```
context Client::connect(h:Host)
 pre: host.isUndefined()
 post: active = true
```

```
inv: hdata < 256
```

```
context Host::send()
 pre: clients->size() > 1
 post: cients->forAll(c|c.data = hdata)
```

```
class Client {
  bool active;
  int data;
  void connect(Host* h) {
    ...
    active = true;
    ...
  }
};
```

(a) UML class diagram with OCL constraints (b) UML object diagram (c) Implementation of Client

Fig. 1. System description with a modeling and a programming language

in a modeling language and an implementation in a high-level programming language, our verification flow proceeds in two stages.

In the first stage, we check the formal specification for behavioral properties which indicate bad states. For instance, we verify that a system is free from deadlocks. For this purpose, the formal specification and its properties are translated to instances of the Satisfiability (SAT) problem. We abstract from the precise implementation of the components using the formal specification as an abstract, behavioral model. The behavioral model is used to detect flaws in the design in the absence of a precise implementation.

Afterwards code representing the structure can automatically be generated from the formal specification, i.e., only the precise implementation is left to the designer. Hence, in the second stage just the correctness of these implementations has to be checked, i.e., we refine the verification task leveraging the implementation and check whether the individual components adhere to the formal specification. Our verification flow allows for an early detection of design flaws and considers the implementation to search for functional bugs only when necessary. To this end, the description of the verification task is automatically generated from the modeling language and implementation language, respectively.

In the following, we use the *Unified Modeling Language* (UML) [10] and the programming language C++ to describe the structure and the behavior of components. Our verification flow, however, applies to other modeling and programming languages, too. Moreover, we restrict our perspective to *finite-state systems*, i.e., the number of components in the system is fixed.

The remainder of the paper is structured as follows. In Section 2, we describe our terminology. In Section 3, we introduce our verification flow and present an implementation using *SAT-based Bounded Model Checking* (BMC) [11]. Section 4 concludes the paper.

2 Preliminaries

We introduce the terminology used in the paper by means of the example shown in Fig. 1: Fig. 1(a) shows a *class diagram* which describes the structure of the classes Host and Client. Additionally, the class diagram is annotated with formal constraints leveraging UML's *Object Constraint Language* (OCL) [12]. These constraints serve as a formal specification. Fig. 1(b) shows an *object diagram* which describes the state of a system consisting of one client and one host component. Lastly, Fig. 1(c) sketches the actual implementation of the client component in C++.

In Fig. 1(a) the structure of all host and all client components is described by *classes*. We call these classes Host and Client, respectively, and we say that the individual host and client components are *instances* (or objects) of the respective classes. A class provides *attributes* and *operations*. The attributes serve as variables which store values. The operations define behavior which can be *invoked* and modify the values of the attributes. For instance, the class Client in Fig. 1(a) has two attributes active and data and one operation connect. The connect operation can be invoked to establish a connection between a client and a host component which modifies the value of the variable active.

The valuation of the attributes of all instances of a system comprise the *state* of the system. For instance, Fig. 1(b) shows a state σ_t, $t \in \mathbb{N}$, of a system, with the assignment $\mathsf{hdata} = 42$, $\mathsf{active} = \mathsf{false}$, and $\mathsf{data} = 0$. When the connect operation is invoked in state σ_t, the attribute active is set to true resulting in a new state σ_{t+1}. We say that there is a *transition* from a state σ_t to another state σ_{t+1} if and only if an operation o exists which results in state σ_{t+1} when invoked in state σ_t. We denote this transition by $\sigma_t \rightarrow_o \sigma_{t+1}$. The object diagram in Fig. 1(b) shows a possible sequence of transitions starting from state σ_t where the connect operation is invoked multiple times.

We use OCL constraints to define formal properties for classes. The OCL constraints are annotated into the class diagram using UML note elements, e.g., in Fig. 1(a). We distinguish three types of OCL constraints: an *invariant* I, a *precondition* $\lhd$, and a *postcondition* $\rhd$. We use $\mathcal{I}$ and $\mathcal{O}$ to denote the sets of all invariants and all operations, respectively. An OCL constraint $\varphi \in \{I, \lhd, \rhd\}$ *holds* in a state σ denoted by $\varphi(\sigma)$ if and only if the respective state σ satisfies the condition φ.

The invariants define global constraints which apply to all instances of a class. Each invariant of a class has to hold for all states. For example, in Fig. 1(a), the value of attribute hdata has to satisfy $\mathsf{hdata} < 256$ for all instances of the class Host in all states.

Preconditions and postconditions define constraints which apply to a particular operation of a class. We use $\lhd_o$ and $\rhd_o$ with subscript o to denote the pre- and post-condition of operation $o \in \mathcal{O}$. The invocation of operation o in a state σ is *valid* if and only if $\lhd_o(\sigma)$ holds.

3 Verification Flow

In this section, we first present the general idea underlying our verification flow in Section 3.1. We then describe the verification tasks of the first and the second stage in Section 3.2 and Section 3.3, respectively. Finally, we discuss implementation details in Section 3.4.

3.1 General Idea

The proposed verification flow is shown in Fig. 2. The inputs of the verification flow are a formal specification described by a UML class diagram annotated with OCL constraints and an implementation of a system written in a high-level programming language. In the figure, we have separated the two stages of the verification flow in two boxes with dashed borders. The top box denotes the first stage which focuses on the class diagram. The bottom box denotes the second stage which uses the implementation to refine the verification task.

The interaction of the components are conducted in the first stage in the absence of an implementation. If the verification on the specification level fails, design bugs can be determined and fixed without considering implementation details. If no further

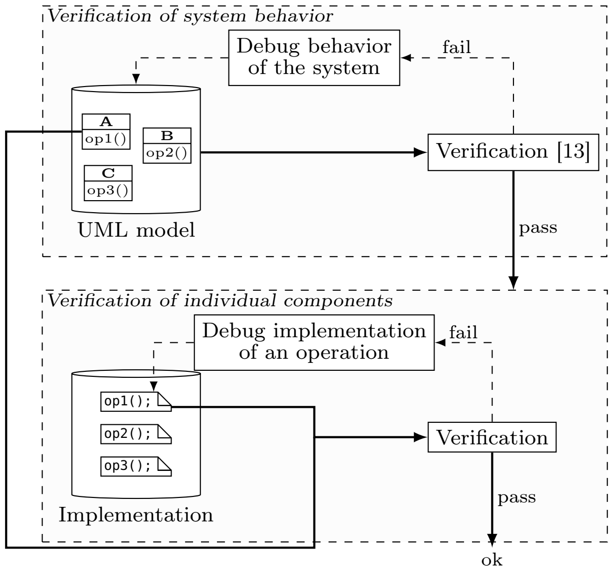

Verification of system behavior
Debug behavior of the system
fail
A
op1()
B
op2()
C
op3()
UML model
Verification [13]
pass
Verification of individual components
Debug implementation of an operation
fail
op1();
op2();
op3();
Implementation
Verification
pass
ok

Fig. 2. Verification flow

design bugs are found, the individual components are verified in the second stage. For this purpose, also information from the specification level is required. This particularly includes the respective pre- and postcondition from the operations as well as the invariants specified in the class diagram. In the second stage, functional bugs are detected which may be caused by an erroneous implementation of operations.

3.2 Verification of System Behavior

In the first stage, we check whether a verification condition τ can be refuted for a fixed system configuration in the absence of an implementation We use the annotated constraints in a UML class diagram as a formal specification. The negated verification condition $\neg\tau$ denotes a formal property which characterizes bad behavior in the system, e.g., a possible deadlock situation.

The system configuration provides a fully or partially defined initial state of the system σ_0, i.e., the number of components is fixed and their attributes are assigned to precise values. We automatically generate a logic formula using a BMC approach. The logic formula encodes a reachability problem on a state-transition graph, where the states correspond to the states of the system and the transitions correspond to the formal specification of the operations. The verification condition defines the states which must not be reachable. A satisfying assignment of the logic formula serves as a counterexample for the verification condition, e.g., a deadlock is a sequence of operation invocations from which the invocation of no other operation is valid.

The generated logic formula for the verification task of the first stage is shown in (1), where $\mathcal{I}$ is the set of invariants.

$$\bigwedge_{t=0}^{k} \bigwedge_{I \in \mathcal{I}} I(\sigma_t) \wedge \bigwedge_{t=0}^{k-1} (\lhd_{o_t}(\sigma_t) \wedge \rhd_{o_t}(\sigma_{t+1})) \wedge \neg\tau(\sigma_k) \qquad (1)$$

The satisfiability of the formula is decided utilizing a state-of-the-art solvers for the SAT problem. From a satisfying assignment, a sequence of operation invocations $o_0, \dots, o_{k-1}$ is derived such that all invariants, pre- and postconditions, and the negated verification condition are satisfied. The length of the sequence of invoked operations is bounded by k, i.e., if no counterexample is found within the bound, k has to be increased, up to a reasonable value.

The verification condition τ can be adjusted for a particular pattern of bad behavior to be checked. For instance, to check for a deadlock we use

$$\tau(\sigma) = \bigvee_{o \in \mathcal{O}} \lhd_o(\sigma), \tag{2}$$

i.e., at least one precondition of any operation has to hold in state σ.

3.3 Verification of Individual Components

In the second stage, we check whether the individual components are functionally correct using the implementation written in a high-level programming language to refine the verification task. Also here, we generate a logic formula for each operation which is satisfiable if and only if the implementation of an operation o does not conform to the formal specification. The logic formula is built by encoding the implementation of an operation as a logic formula impl_o conjoined with the precondition, the postcondition, and the invariants.

We consider two states σ_t and σ_{t+1} denoting the system before and after the operation o has been invoked. Furthermore we assume that the invariants are correct, i.e. our approach considers an underapproximation of the state space if the invariants are too strict. The problem whether the implementation of the operation is functionally correct, i.e., the implementation conforms to the specification, is formalized as (3):

$$\bigwedge_{I \in \mathcal{I}} (I(\sigma_t) \wedge I(\sigma_{t+1})) \wedge \lhd_o(\sigma_t) \wedge \text{impl}_o(\sigma_t, \sigma_{t+1}) \wedge \neg \rhd_o (\sigma_{t+1}) \tag{3}$$

The logic formula checks for the existence of two states σ_t and σ_{t+1} such that σ_t and σ_{t+1} satisfy the system invariants $\mathcal{I}$, σ_i satisfies the precondition $\lhd_o$, σ_{t+1} violates the postcondition $\rhd_o$, and impl_o describes the behavior of the operation's implementation, i.e., the transition $\sigma_t \rightarrow_o \sigma_{t+1}$ with o. A satisfying assignment corresponds to a counterexample which proves an inconsistency of the formal specification and the implementation. The counterexample can be used to analyze and fix the inconsistency. Otherwise, if the logic formula is unsatisfiable, no such counterexample exists, i.e., the operation's implementation conforms to the formal specification.

3.4 Prototypical Tool

We have built a prototypical tool which implements the described verification flow. This tool generates a logic formula representing the considered verification tasks and utilizes a *Satisfiability Modulo Theories* (SMT) solver to determine a solution for it. The generation of the corresponding SMT solver input from the OCL constraints is described in [14, 13]. To generate the logic formula from the C++ source code, we leverage the *Low Level Virtual Machine* (LLVM) [15] compiler infrastructure as described in [16]. First, we translate C++ to an *LLVM Intermediate Representation* (LLVM-IR). Then, we use a BMC approach [11] to translate the LLVM-IR into a logic formula. We unroll loops in the LLVM-IR for a fixed number of iterations, introduce one logic variable each time a program variable is written, and encode the individual instructions to semantically equivalent logic constraints. Finally, we conjoin the logic formula encoding the OCL constraints and the logic formula encoding the implementation and assert correlating variables in the resulting logic formula to be equal.

4 Conclusions

We proposed a two-staged verification flow which focuses on the ESL. The verification flow is applicable to a system which is formally specified in a modeling language and implemented in a high-level programming language. In the first stage, we use light-weight model checking to detect design bugs early without considering the implementation details. Thus, the first stage can be used even if the implementation is not available. In the second stage, we refine the verification task leveraging the implementation to detect functional bugs. In both stages, the description of the verification task for the solving engine is automatically generated.

References

1. H. Foster, "Applied Assertion-Based Verification: An Industry Perspective," *Foundations and Trends in Electronic Design Automation*, vol. 3, no. 1, pp. 1–95, 2009.
2. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking.* Cambridge, MA, USA: MIT Press, 1999.
3. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Tools and Algorithms for Construction and Analysis of Systems.* Springer, Mar. 1999, pp. 193–207.
4. M. Y. Vardi, "Formal Techniques for SystemC Verification; Position Paper," in *Design Automation Conference.* IEEE, June 2007, pp. 188–192.
5. B. Bailey and G. Martin, *ESL Models and their Application: Electronic System Level Design and Verification in Practice.* Dordrecht, Heidelberg, London, New York: Springer, Dec. 2009.
6. F. Rogin and R. Drechsler, *Debugging at the Electronic System Level.* Dordrecht, Heidelberg, London, New York: Springer, July 2010.
7. T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Decomposing Refinement Proofs Using Assume-Guarantee Reasoning," in *Int'l Conf. on Computer-Aided Design.* IEEE, Nov. 2000, pp. 245–252.
8. R. P. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, "Specify, Compile, Run: Hardware from PSL," in *Int'l Workshop on Compiler Optimization Meets Compiler Verification*, Mar. 2007, pp. 6–20.
9. ——, "Automatic hardware synthesis from specifications: a case study," in *Design, Automation and Test in Europe.* ACM, Apr. 2007, pp. 1188–1193.
10. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual.* Essex, UK: Addison-Wesley Longman, Jan. 1999.
11. E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Tools and Algorithms for Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 2988. Springer, Mar. 2004, pp. 168–176.
12. J. Warmer and A. Kleppe, *The Object Constraint Language: Precise modeling with UML.* Boston, MA, USA: Addison-Wesley Longman, Mar. 1999.
13. M. Soeken, R. Wille, and R. Drechsler, "Verifying Dynamic Aspects of UML Models," in *Design, Automation and Test in Europe*, Mar. 2011, pp. 1077–1082.
14. M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using Boolean satisfiability," in *Design, Automation and Test in Europe*, Mar. 2010, pp. 1341–1344.
15. C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Int'l Symp. on Code Generation and Optimization.* IEEE Computer Society, Mar. 2004, pp. 75–88.
16. H. Riener and G. Fey, "FAuST: A Framework for Formal Verification, Automated Debugging, and Software Test Generation," in *SPIN Workshop*, July 2012, pp. 234–240.