User-friendly Model Checking Integration in Model-based Development *

Alarico Campetelli, Florian Hölzl, Philipp Neubeck
Institut für Informatik
Technische Universität München
Garching bei München, 85748, Germany
{campetel, hoelzlf, neubeck}@in.tum.de

Abstract
We present our approach to a user-friendly model checking integration in model-based development. The used modeling tool is AutoFocus 3, developed at our research group and specialized for reactive and embedded systems. For this integration, we approach usability at four points: tight coupling of verification properties with model elements, different specification languages for the formulation of properties, visualization of counterexamples as well as evaluation of different model checkers for adequate performance. Dealing with these issues leads to one of the first model-based development environments incorporating property specification, model checking and debugging.

Keywords: verification, model checking, model-based development, tool support, embedded systems

1 Introduction

With an increasing number of software-intensive systems and their steady increase in size and complexity, verification of their correct operation is critical. Formal verification based on formal methods is a technique to prove this correctness in an exhaustive manner, compared to testing which is partial in principle. Still, the developers' acceptance of formal verification methods is rather limited. Be it because of bad tool integration, unacceptable tool performance or lack of knowledge. Formal verification techniques should be supported and applied throughout the whole development process, but currently they are applied only in rare cases or are abandoned completely. This calls for integrated development environments, where design and verification tasks go hand in hand.

A model-based development environment provides model types to design different aspects of a system, e.g., for embedded systems its functional behavior, its software architecture and its hardware architecture. These models are used to automatically generate the implementation code of the software system w.r.t. its hardware and execution environment. The design and code generation capability of the development environment is complemented by validation and verification mechanisms, such as test case generation, model checking and theorem proving.

AutoFocus 3 [11] is a scientific tool for the development of reactive, software-intensive, embedded systems based on the semantics of the Focus modeling theory [5] and implemented on top of the Eclipse¹ platform. In this tool, systems are modeled as software architectures based on components combined with executable behavior descriptions (e.g., automata with input and output). Hardware aspects are captured in a topology model, which describes execution and transmission units such as electronic control units and bus systems. A deployment model allocates components to execution units and allows to generate the complete code of the system. From the validation point of view, AutoFocus 3 provides a simulation environment, a test case generator as well as the model checker integration. The advantages of model checking techniques suit with the criticality of embedded systems development, which require exhaustive verification. Furthermore, an on-target debugging and co-simulation mechanism is available.

The current paper presents the actual state of the model checking integration into the AutoFocus 3 development tool and explains the underlying approach we are following. We tackle usability of verification mainly in four areas. Firstly, we integrate specification and verification tightly into the model-based development process. This means that verification properties are linked to model elements and can be verified locally and easily during the development, for details see Sec. 2. The support for different specification lan-
The scope of a verification property is fixed by the component it is attached to. Thus, the property can refer to the i/o-ports of this component. If available, the property can additionally refer to all i/o ports of the immediate subcomponents. If the component is defined by an automaton, the automaton’s state variables are also in scope.

After augmenting the model with verification properties, we are able to choose the component to verify. If we verify a component, this component together with all its subcomponents are considered. The chosen component is verified against all possible inputs allowed by the types of its input ports, thus the components environment (consisting of the non-descendant components) is completely ignored. Contrary, if we would like to verify a component together with its sibling components, we would have to verify the parent component containing both. Thus, the chosen component fixes the verification context.

For illustration, Fig. 1 shows a small component network in AutoFocus 3. In this view, the user selects a model element in the upper part and a predefined property template from the list below. The templates contain free variables which the user has to define. Therefore, he can refer to model elements like i/o-ports or state variables. The context of the verification is selectable from a drop-down list containing all parent components. Finally, the verification can be initiated. The verification is performed by a model checker. If the model checker is able to prove the property, the user is presented a confirmation. If the property is disproved, a counterexample is generated for the user’s review. Counterexamples can be simulated or illustrated as MSCs, see Fig. 2 and 3.
Benefits As explained, the user specifies a verification condition in the scope of a certain component. This allows the tool to facilitate the formulation of verification properties which would not be possible otherwise. For example, the property’s formula is type checked with respect to the types of the ports and state variables. Other supportive functionality is discussed in more detail in Section 3. This shows that the tight coupling of properties to the system model improves the process of verification and validation.

Our approach also allows to locally verify properties of a component. By verifying components and their subcomponents independently from the surrounding environment, the developer can apply verification continuously throughout the development. Thus, implementation errors can be found earlier and bug-fixing costs are reduced.

Contrary to operational behavior descriptions, the developer can formulate verification properties in a declarative way. In many cases, a declarative description is considerably shorter and comprehensible than the operational description. Furthermore, each property is specific to a certain aspect of the component’s functionality allowing to abstract from the complexity of the component’s behavior. With the presented tool integration, the developer can easily formulate such properties during the component’s development.

Similar the unit tests, which should be defined together with the implementation, we think that our model checker integration allows the same approach using verification. This way, as the system development proceeds, several verification properties are collected and contribute to the confidence in the system’s implementation.

3 Specification of Properties

The formal properties to be verified on the system model are specified in temporal logic, Computational Tree Logic (CTL) and Linear Time Logic (LTL) for SMV and μ-calculus for TVARC. Practical application of such temporal logics requires advanced skills in formal verification, since it is difficult to express properties in these formalisms. The acquisition of this level of knowledge may be an obstacle to the adoption of automated verification tools. In order to improve the usability and expressiveness in the specification of the properties, we propose three higher level approaches. The most intuitive is based on predefined Property Templates, the second uses a pattern-based approach, and the last is based on the Structured Assertion Language for Temporal Logic (SALT) [2].

The first approach, as highlighted in Fig. 1, provides templates that can be directly selected, configured and executed from the graphical interface. We represent common properties to be verified, which are integrated in the model checker view and can be saved along with the model itself. The user selects an element in the model view, then the appropriate templates are shown. Examples of property templates for a port and a component are, respectively:

- Port value is eventually equal to X
- After input port A has value X, output port B has eventually value Y

Templates contain variables, e.g., “X” in the precedent templates. Each variable is bound to the content of an associated text field. Using these text fields, the user defines logical parts of the formula. Variables correspond to states or port values or, if they are indicated as “events”, to general formulas. A type and name check is made for controlling the correctness of the values inserted by the user.

The second approach is a pattern-based approach for the presentation, codification and reuse of property specifications for verification. We refer to the specification pattern definition given by [8]: “A property specification pattern is a generalized description of a commonly occurring requirement on the permissible state/event sequences in a finite-state model of a system. A property specification pattern describes the essential structure of some aspect of a system’s behavior and provides expressions of this behavior in a range of common formalisms”. The patterns are based on a survey of available specifications, collecting over 500 examples of property specifications. Most of these specifications were instances of the patterns.

We added specification patterns in the model checker view, where the user has to select from a list
4 Handling of Counterexamples

In case the result of a verification is negative, the counterexample generated by SMV is reported in a tool view. The counterexample represents a path in the SMV model. This path can lead from the initial state to a state where the property is violated or it can contain a loop, where the desired behavior is never reached. The path is visualized as a sequence of states, together with the eventual loop information. In this sequence each state describes current states of the state machines and a valuation of all variables and i/o ports that compose the model.

In order to improve the representation and understanding of such data, we perform the visualization of the counterexample as a message sequence chart. In the generated diagram, the actors are the different components of the model and the messages between them represent the value assignment in the model from an output port of the sender component to the input port of the receiver component. It is also possible to perform the simulation of the counterexample in AutoFocus 3. The view shown in Fig. 3 contains controls to execute forward and backward steps as well as continuous playback of the simulation. In other views the user can examine the current values of i/o ports and a state machine’s transition system with the highlighted current state. A step in the simulation corresponds to the passage from a state to the following one in the SMV counterexample.

In case a property is disproved during the verification, these views help to debug the model. The described debugging phase is solely based on the structures previously modeled by the user. Thus, the intermediate model generated for the model checker is de facto bypassed. We consider the integration of the verification and counterexample analysis in the tool to be a big advantage for the user as he is not bothered with the underlying structures and the model checker anymore.

5 Implementation

For the usability of verification it is essential to reduce delays between writing verification properties and their verification. The user must be able to iteratively adapt the properties and the system model. This is impossible if a single verification takes several hours of computation. Therefore we evaluate several model checkers which implement different optimization strategies. Currently, we have implemented the translation of the model to the SMV and TVARC model checkers.

We apply the TVARC model checker to the C0 code generated from the model, to be precise its C intermediate language representation, and SMV directly to the model design, as depicted in Fig. 4. Cadence SMV is a well-known model checker, branched...
from the original SMV and developed by the Cadence Berkeley Labs. We generate two artifacts from the system model, namely a C0 (a subset of language C) code implementation, and an SMV representation. The correctness of the transformation to C0 has been shown in [10]. The correctness proof of the export to the SMV model has not been published yet.

For the execution of the SMV model checker, AutoFocus 3 automatically translates the selected component with all its subcomponents into an SMV instance, where each state machine is represented by an SMV module. Each module refers to the modules of the relative subcomponents.

During this translation, we have restrictions for recursive constructs. In general, AutoFocus 3 supports recursive functions and data types. However, the SMV generator does not permit them, neither does the C0 generator. This limitation is due to the finiteness of the state space (required by SMV and often convenient in embedded systems design) and the fact that recursive data types induce a potentially infinite state space (of course, only a finite part may be reached). In such a case, we return an error message to the user.

Another limitation is given by large state variables like integer variables. Excessive use of these quickly leads to the so-called state explosion. Even in small projects, we noticed that the verification process required too much time and memory. Therefore, in the model checker view, the user can restrict the value range of integer variables for the whole model, and also define a specific range for certain variables (Fig. 1). It is important that the user specifies the smallest possible integer interval to avoid the state explosion problem. With static analysis, we could automatically determine these intervals.

The application of abstraction techniques may help to reduce these problems. TVARC supports such abstractions, which is beneficial in our preliminary experiments. TVARC abstracts the model using abstract interpretation, that has the advantage of reducing the complexity of the original data structures involved in the verification. Moreover, it uses the more expressive temporal logic $\mu$-calculus. If the property or model to verify does not fit with model abstraction applied from TVARC, SMV may be faster for the same model, because it applies a different verification technique. We expect that TVARC works better for models where the data types are complex. In contrast using SMV, models with large integer intervals result in a huge state space rendering the verification infeasible.

Although the correctness of the generators can be shown mathematically, the generators may contain implementation faults. Since this is an inherent problem of code generators and formal verification in general, we propose to tackle it by methodical steps. We can additionally verify the generated C0 implementation with TVARC, using the same properties as for the SMV model checker and compare the results.

6 Related Work

To the best knowledge of the authors, an integrated and user-friendly verification tool environment as presented in this paper has not been realized before. Anyway, some modeling tools integrate formal verification capabilities. Simulink Design Verifier and SCADE Design Verifier are extensions for Simulink and the SCADE tool, which use formal analysis techniques provided by Prover Plug-In. Design Verifier can only check assertions and not temporal logic properties as our model checking approach allows.

Rhapsody in C++ [13] has a verification environment for UML models, the specifications to be verified can be formulated using temporal patterns or a graphical specification formalism called Life Sequence Charts [7]. The verification is applied to UML models composed by UML state charts and UML class diagrams. They are transformed in a format for VIS model checker [4]; a finite state machine for the model and a temporal logic CTL formula for the property to verify. In Hugo/RT [1] UML state chart diagrams are associated with a UML class diagram to specify the model to verify and communication diagrams describe how the objects of a model may interact. The tool generates input for two model checkers: SPIN and UPPAAL [3]. The model checkers can verify whether the interactions expressed by a UML communication diagram are realized by the state machines. Considering the specification of the model checking properties, none of these approaches integrates together user-friendly templates with correctness checks, specification patterns, assertion language and temporal logic. Additionally, none of these tools has an integrated counterexample handling and simulation.

7 Conclusion and Future Work

The current paper presented one of the first holistic and user-friendly verification environments with a spotlight on the model checker integration. We support two different model checkers for the model and implementation code verification. In order to enhance the usability of model checking in model-based development, we integrate in the modeling tool different property specification concepts. We support specification patterns, an assertion language, and property specification based on templates adapted for the model artifacts. Moreover, we include back-translation of counterexamples to the model level, e.g., counterexam-
ples can be examined step by step in the AutoFocus 3 simulator. Our idea is to render model checking usable not only by skilled developers, but also by common users. Moreover we think, that the strong integration in model-based development is an advantage.

SMV has already been tested in a case study from the automotive domain [9], whereas the testing phase of TVARC’s integration is just being finished. We consider as future work the extension of the TVARC integration with the properties paradigms already implemented for SMV and the direct translation from system models to TVARC’s model representation.

We plan to conduct more case studies with industrial partners involved in the automotive and embedded system domain. We will focus on verification and domain-specific property specification. We will analyze the performance of the model checkers in terms of memory and time, also to enhance TVARC. Due to a better model abstraction, we expect TVARC’s results to be better than those of SMV, especially when the model has redundant data that does not affect the validity of the property to verify. Since the performance of verification may still be infeasible for some model instances, we consider to introduce a timeout concept and bounded model checking as a fallback method.

Acknowledgements

We would like to thank Daniel Ratiu and Fortiss GmbH in Munich for the support in the tool integration.

References


