Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 391–399.

On the Verification Problems of the Component-Based Software Development*

Zsolt Borsi[a], László Kozma[a], Anna Medve[b]

[a]Department of Software Technology, Eötvös Loránd University

[b]Department of Electrical Engineering and Information Systems, University of Pannonia

Abstract

The verification problems of the CBSD are related to the fact that a component must be correct with respect to its specifications respectively that the entire system must be correct with respect to its requirements. In this paper, the relation of testing and model checking methods are discussed. An example is given to demonstrate that model checkers can be successfully used for validating usage models and supporting model-based testing activities.

Keywords: component-based software development, model-based testing, model checking

1. Introduction

This paper focuses on the verification problems of the Component-Based Software Development (CBSD) using a CBSD method (e.g. KobrA). CBSD methods constitute a framework for supporting the entire development process and guiding all the activities from specification to validation by providing methodologies and tools to the developer making his decision easier when, where and how to do. The KobrA method permits of decomposing the entire system into finer-grained parts and mapping them to existing functionality of the system [1]. The method suggests a development process according to a three-dimensional model (composition/decomposition, abstraction/concretisation, genericity/specialization) and it supports to handle verification problems of the CBSD process as well [2]. Model checking tools enable to design complex systems with considerable assurance regarding the correctness properties. The results of model-based testing researches provide methods for modelling testing structure and test behaviour to obtain correct and robust development artifacts. In this paper an example is given presenting that model checking can be used to prove that the usage model of the component is

*This research work was supported by TÁMOP-4.2.1/B-09/1/KMR-2010-0003.

valid against the user's requirements. This fact demonstrates that model checking is a convenient way for extending model based methods during high level abstraction of modelling.

Section 2 summarizes the basic concepts and properties of components. Section 3 reviews the component-based software development activities which are related to model-based testing and model checking. The latter are introduced shortly in Section 4. Section 5 presents a verification example. Section 6 summarizes our approach and describes plans for future work.

2. Component-Based Software Development (CBSD)

The component-based software development (CBSD) consists of two main activities: the component engineering and the application engineering. The first one deals with how individual components need to be built to be reusable and the latter one addresses the assembly and integration of the building blocks into a new software system.

The component is a central notion of CBSD. This fundamental building block is a reusable unit for composition. According to many definitions it becomes obvious that components are basically built on the same fundamental principles as object technology [1, 2, 3] since an object can be viewed as a special component.

Partitioning a design into components is a subtle process that has a large impact on the success of the resulted components. It has two aspects – the requirements and the catalogue of available components. Szyperski in [3] argued that relevant aspects governing granularity demand fine coarse-grained partitioning among services of the system.

All components have two interfaces: the provided and the required interface. Interfaces are the only means for accessing the services of a component comprising operations provided or expected in terms of services. The operations are the access points between components and their functionalities depending on pre- and postconditions that constraint invocation and termination validity for the given operation. A state is a distinct combination of a component's internal attribute values that are constantly changed through the operations and that govern the possibility for an operation to be invoked under the conditions of a certain state. The implementation of a component realizes the private design of the component. It is hidden inside the encapsulating shell of the component and is arbitrarily exchangeable through any other implementation that realizes the same external features.

The UML can be used at a higher level of abstraction to model and specify system artifacts including architecture, functionality of components and collaboration between entities of a system.

3. Verification Problems on the Component-Based Software Development

The verification problems have shown up on two levels of the CBSD: on the level of component and on the level of entire system. Verifying that a component is correct with respect to its specifications and verifying that the entire system is correct with respect to its requirements, is a task of CBSD. Verification methods are: correctness proofs, synthesis, testing, model checking.

Software testing is a widely used and accepted approach for verification and validation of a software system. In general, testing involves three main tasks: test suite definition, test execution and test analysis. The main limitation of testing is that it can only be performed within or after the implementation activity. As a consequence, modular checking is the way for the analysis of component properties based only on the component and its interfaces.

The combination of modelling and testing can be represented by two orthogonal dimensions namely model-based testing and test modelling. Model-based testing is the development of testing artifacts on the basis of UML models. The testing and model checking complement each other in practical use and contribute to improve test modelling. Test modelling concentrates on how to model testing structure and test behaviour with the UML.

KobrA supports to solve verification problems of the CBSD process. In [1] incremental development approaches have been introduced for the KobrA method. In order to perform test suite generation activity, in KobrA method it is possible to define test cases during the component specification and realization activities [2]. Testing is applied to generate modes of operation on the final product that show whether it is conforming to its original requirements specification, and to support the confidence in its safe and correct operation. A textual use case description has for usage modelling by indentified roles and entities, and for interaction modeling by identified operation. Three main artifacts are used from use-case literature: use case diagrams, textual use case specifications and sequence diagrams. In Section 5 an example is given and discussed to demonstrate that model checkers can be successfully used for validating usage models and supporting model-based testing activities.

4. Model checking

Model checking is a verification technology testing a finite representation model of the system against a set of requirements [4, 5, 6, 7, 8]. In order to check whether the system satisfies the expected properties which constitute its specification, either the system design and the properties has to be modelled formally. A model checker tool provides an algorithmic mean determining whether the given abstract model satisfies its specification expressed by a set of temporal logic formulas. The model checker tools usually support the temporal logic languages Linear Temporal Logic

(LTL) and Computational Tree Logic (CTL). During the verification procedure the tool achieves an automated search analyzing the state space of the model and investigates every possible behaviour of the system it represents. Then the tool informs the user whether the properties are true or not. If the model checker detects violation, most tools provide the user with a counter-example. A counter-example represents a possible scenario of the system in the form of a sequence informing the user about the source of the error. Using model checking you can not avoid the major problem of this domain called state explosion: exploring the complete state space often leads to running out time and available memory. After all this technique proved to be successful and efficient for verifying automatically large, complex systems in practice.

5. A verification example

Given the use case description of PurchaseItem from the usage model of the vending machine introduced by Gross [2]. This use case represents all the activities the user may perform buying an item from the machine. It is the specification of what a vending machine supposed to do supporting the buying course.

In this example only an excerpt of the above use case description is used. On this high level of abstraction the PurchaseItem functionality does not consist of dispensing change or checking the amount of money. The simplified functionality of the vending machine comprises the activities like inserting coins, selecting the item that we would like to buy and taking the paid item. Besides the user can abort the buying process and the machine is waiting for a user when there is nobody using its services. Table 1 shows the steps of the only buying process our vending machine accepts.

1.	User inserts sufficient amount of money.
2.	User selects an item on panel or abort the buying course
3.	The selected item is dispensed.
4.	User takes item.

Table 1: The excerpt of the use case description of PurchaseItem

In the following two task will be investigated which usually have to be solved when setting up a model checking process. First, the model of the system design will be constructed and described formally in the input language of the NuSMV model checker [6]. The next step is identifying and formalizing the properties we are interested in and want to check. The specification of a system can be determined as a set of temporal logic formulas. In this example we represented the specification by CTL formulas. Either the specification and the model of the vending machine has to rely on nothing else but the information we can retrieve from the use case description.

The system supports or perform the following activities: insertcoin, selectitem,

takeitem, abort, idle. In every state through its execution these are the only activities the vending machine is allowed to perform. Figure 1. shows from which part of the use case description the parts of NuSMV model and specification can be derived.

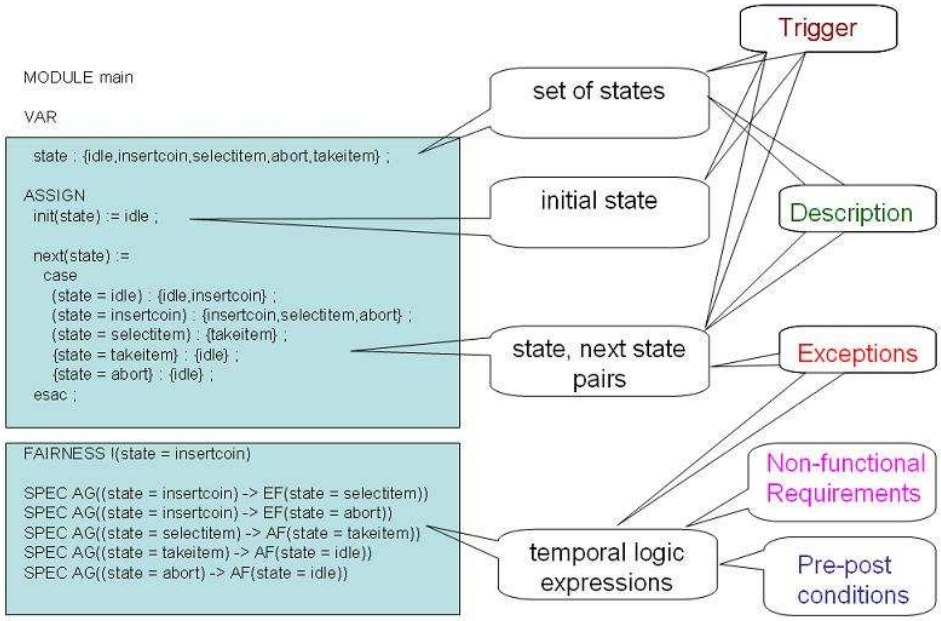

Figure 1: Mapping the corresponding parts of the use case description to parts of the NuSMV model and specification

The model specification in NuSMV consists of the possible values of variables which determine the space of states, the initial values of the variables and the transition relation as the input language of NuSMV is designed for describing Finite State Machines. The transition relation is expressed by pairs defining the value of variables in the next state depending the value of variables in the current state. In this example the variable state can take the values idle, insertcoin, selectitem, abort, takeitem which imply the possible activities of the vending machine. We can deduce the initial states from the Trigger part of the use case description, while the Trigger, Exception and Description parts imply the elements of the state set of the vending machine. The constrains in the Pre- and Postconditions and in the Non-functional Requirements can be mapped into temporal logic expressions.

NuSMV offers features for generating possible executions (so called traces) of a model described in the NuSMV input language. Traces can be obtained automatically or can be built interactively asking the user in every step to to choose the next state from a set of possible successors. By running command `simulate -pr 10` the system generates a ten-steps trace in which the the next states are picked

randomly in every step. In Figure 2 the output produced by NuSMV represents a trace where the vending machine repeatedly enters the states insertcoin, selectitem, takeitem and then goes to the initial idle state.

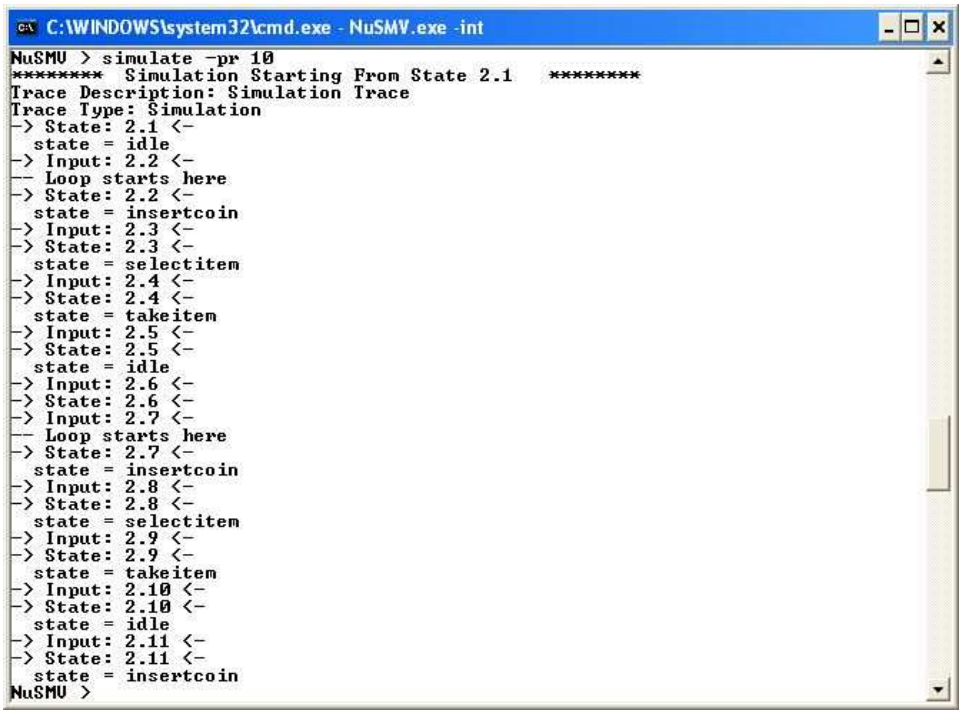

Figure 2: A possible execution of the model representing a possible behaviour of the vending machine

Consider the specification below:

(1) FAIRNESS !(state = insertcoin)
(2) SPEC AG((state = insertcoin) -> AF(state = selectitem))
(3) SPEC AG((state = insertcoin) -> EF(state = abort))
(4) SPEC AG((state = selectitem) -> AF(state = takeitem))
(5) SPEC AG((state = takeitem) -> AF(state = idle))
(6) SPEC AG((state = abort) -> AF(state = idle))

Table 2: The requirements are formalised in CTL

The requirements are expressed in terms as invariants and the single properties are numbered to make the following discussion about them easier. Property (1) expresses the natural expectation that it can not be true that the machine is in the insertcoin state in infinitely many cases. Property (2) asserts that it is always true

that all executions containing the state insertcoin will lead to the state selectitem. Property (3) claims that for all states of all traces if the property state = insertcoin holds then exists a path which leads to a state where `state = abort` holds.

As our requirements are expressed and the model of the vending machine is constructed, verifying the model against its specification can be done using NuSMV model checker. The tool in Figure 3 report us that properties (3)–(6) proved to be true but on the other hand property (2) is not true. NuSMV informs us by generating a counter-example, that is a sequence of states that exhibits a valid behaviour of the model that doesnt satisfy the specification: the sequence begins in the idle state and repeats the insertcoin-abort-idle states in this order forever.

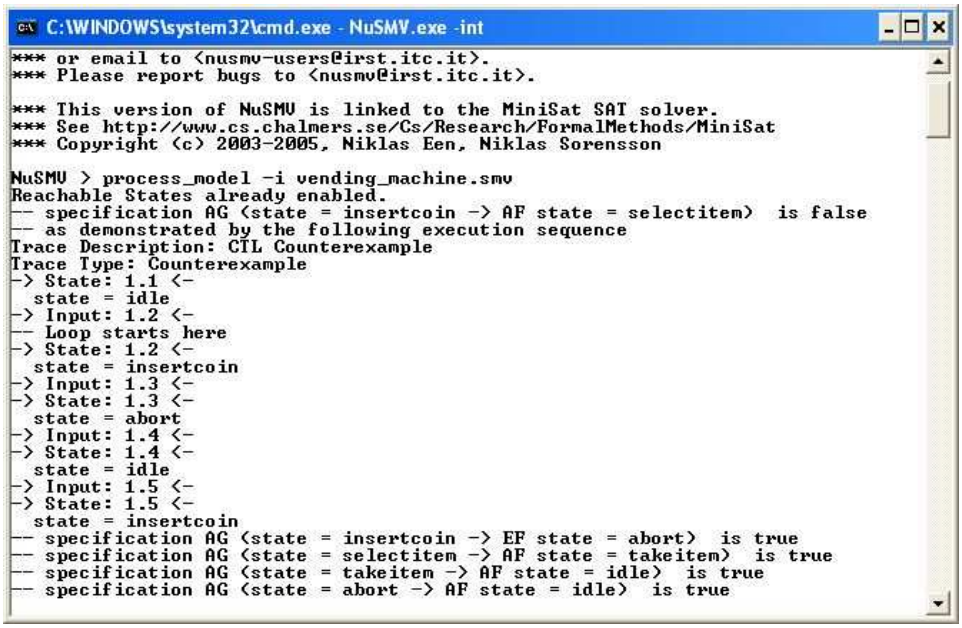

Figure 3: A counter-example produced by NuSMV

One can say that the model of the vending machine is not correct. This statement is not meaningful. The only thing we can declare that the model of the system and the specification does not match. Correctness is a relative notation, assertions like "the system is not correct" only make sense with respect to a statement, to the specification of the system. Actually in this case property (2) was too rigourous omitting the possible behaviour that the buying process can be aborted. In fact in the first attempt we deliberately expressed a specification which can be violated. Our goal was to force the model checker to produce a counter-example which of course does not persuade the costumer that the system performance is sufficient. On the contrary this reported discrepancy exposing a case, a possible scenario in which the implementation violates the users's intent.

In our improved attempt we consider the same system, but modify property (2)

and keep everything (including the model and other properties) unchanged:

```
FAIRNESS !(state = insertcoin)
SPEC AG((state = insertcoin) -> EF(state = selectitem))
SPEC AG((state = insertcoin) -> EF(state = abort))
SPEC AG((state = selectitem) -> AF(state = takeitem))
SPEC AG((state = takeitem) -> AF(state = idle))
SPEC AG((state = abort) -> AF(state = idle))
```

In this case the model checker tell us that all the properties are true, which means the model satisfies the desired specification. The model is correct to the PurchaseItem's specification.

In our approach both the model and the specification were built manually from the given use case description. At the low level of abstraction the counter-examples produced by NuSMV model checker can be difficult to understand as the model excludes the irrelevant details. In general understanding the row output from the NuSMV needs knowledge about traces and the model itself which an end user not necessarily has. A research between the University of Queensland and Queensland Rail[9] used model checking for the verification of railway interlocking designs. Their approach derived the NuSMV model and specification automatically from the system design and the requirements which were given in standard formats. The propoerties they checked were avoidence of train derailments and collisions.

6. Conclusion and future work

This paper has described an approach for verifying the usage model using the NuSMV model checker. It has been pointed out that model checking can significantly contribute to the early detection of faults and errors in the early design phases. At the level of use case model it is still not decided which functionality of the system will be realised in software and in hardware. In this way verifying the usage model can be useful for determining errors even in the hardware.

There are a number of issues we plan to investigate in the near future. First, we will move toward more concrete representation and translate the abstract models in the input language of the NSMV tool in order to verify them. Second, we plan to use model checker to retrieve test sequence generations from the various models.

References

[1] C. Atkinson et al. *Component-Based Product-Line Engineering with UML.* Addison-Wesley, London, 2002.

[2] Haus-Gerhard Gross. *Component-Based Software Testing with UML.* Springer-Verlag Berlin Heidelberg, 2005.

[3] C. Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, London, second edition, 2002.

[4] E.M. Clark Jr., O. Grumberg and D.A. Peled. *Model Checking.* The MIT Press, Cambridge 2000

[5] Bucchiarone, H. Muccini, P. Pellicione, and P. Pierini. *Model-Checking plus Testing: from Software Architecture Analysis to Code Testing.* Lecture Notes in Computer Science, LNCS, vol. 3236, pp. 351–365, 2004.

[6] NuSMV Model Checker `http://nusmv.irst.itc.it`

[7] Ákos Dávid,- László Kozma. *Educational aspects of incremental model checking.* in Proceedings of the 3rd International Multi-Conference on Society, Cybernetics and Informatics, Vol 2, pp. 190-194, 10-13, 2009, Orlando, Florida, USA, ISBN-10: 1-934272-73-6, ISBN-13: 978-1-934272-73-2., `http://www.iiis.org/CDs2008/CD2009SCI/EISTA2009/index.asp?id=0\&area=4`

[8] Kupferman, O., Vardi, M.Y., Wolper, P. *Module checking.* Information and Computation 164(2), 322–344, 2001.

[9] L. van den Berg, P. Strooper, W. Johnson. *An Automated Approach for the Interpretation of Counter-Examples.* ENTCS 174 (2007) 19–35.

Zsolt Borsi, László Kozma
Eötvös Loránd University
1117 Budapest
Pázmány Péter sétány 1/C.
Hungary
e-mail:
`bzsr@inf.elte.hu`
`kozma@ludens.elte.hu`

Anna Medve
University of Pannonia
8200 Veszprém
Egyetem u. 10.
Hungary
e-mail: `medve@almos.vein.hu`