periodica polytechnica

Electrical Engineering
51/3-4 (2007) 133–139
doi: 10.3311/pp.ee.2007-3-4.07
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

Extending a system with verified components

Ákos Dávid / *Tamás* Pozsgai / *László* Kozma

Received 2007-10-03

Abstract

The verification of component-based systems can be extremely complicated because it is usually not possible for system developers to pre-check the compatibility of the individual parts before the actual integration takes place.

A system cannot be considered correct if its components do not work properly. Unfortunately, all the information on the correctness of the individual components become irrelevant and out-of-date from the moment they are used anywhere but the original environment. The solution to this problem can be based on the idea of building correct programs in which reliability is built-in.

In this paper open incremental model checking – addressing the changes to a system rather than re-checking the entire system model including the new extensions – is discussed and compared to traditional modular model checking methods. In our paper we study the practical aspects and the efficiency of using Open Incremental Model Checking by working out a sample system consisting of verified components.

Keywords

model · component · model checking

Acknowledgement

This research work was supported by GVOP-3.2.2-2004-07-005/3.0.

Ákos Dávid
Department of Mathematics and Computing, University of Pannonia,, 8200 Veszprém, Egyetem u. 10., Hungary
e-mail: davida@almos.uni-pannon.hu

Tamás Pozsgai
Department of Mathematics and Computing, University of Pannonia,, 8200 Veszprém, Egyetem u. 10., Hungary
e-mail: pozsgai@szt.vein.hu

László Kozma
Department of Software Technology and Methodology, ELTE, 1117 Budapest, Pázmány P. sétány 1/c., Hungary
e-mail: kozma@ludens.elte.hu

1 Introduction

One of today's most widely used software technology methods is component-based software development (CBSD). It builds around two basic concepts [6, 11, 14, 15, 22]:

– Reuse means the use of preexisting software with or without some modifications.

– Evolution keeps the costs of a highly componentized system low by enabling the replacement of certain components without affecting the functionality of other parts.

There are three conditions to be satisfied in order to use these driving forces of reuse and evolution for developing component-based systems.

Component library Reuse is only applicable if there are existing software components to be used for a specific problem domain. The set of available components should be organized into a component library, especially for educational purposes.

Component model A component model is also needed to support the composition of an application based on standards. The most generally used component models in the commercial area are CCM (CORBA Component Model), COM+/.NET (Microsoft Component Object Model Plus) and EJB (Enterprise Java Beans).

Software architecture Today's more and more complex systems cannot be built without a backbone to support the design and development process, which is software architecture. As there are several definitions for it, the authors consider software architecture an abstraction of a system that defines all the software elements, their properties and the relationship between them [2].

In the future we become more and more dependent of the proper functioning of computer systems, so our confidence in the correctness of such systems needs to be increased. Today's most widely used techniques of testing are only able to post-verify applications and they are not able to guarantee that there are

no more hidden errors left in the design or in the code. However, with the emergence of formal methods, it became possible to create correct programs with respect to a given specification. Different types of synthesis methods, correctness proof and model checking are the most widespread formal techniques in practice today.

Section 2 gives an overview of today's most applicable formal verification method, model checking. Section 3 presents the drawbacks of component-based software development concerning the verification of such systems, and mentions some of the possible solutions with a special emphasis on contracts. In Section 4 a sample system is presented. Section 5 shows the verification of the components using standard model checking. Section 6 discusses the method of open incremental model checking. Section 7 details the future work while Section 8 presents some concluding remarks.

2 Model checking

Model checking is an automatic technique for verifying finite state concurrent systems [5, 12]. It has a number of advantages over traditional approaches. It is sufficient for the user to provide a high level representation of the model and the specification to be checked, so no verification expert is needed. The procedure normally uses an exhaustive search of the space of a system to either terminate with the answer true, indicating that the model satisfies the specification, or give a counterexample that may give an important clue in finding subtle errors in complex systems. The procedure is also quite fast, even on moderate-sized computers. In some cases infinite systems may be verified using model checking in combination with various abstraction and induction principles. Finally, the logic used for specifications can directly express many of the properties that are needed for reasoning about concurrent systems. Temporal Logic makes it possible to describe how the behaviour of such systems evolves over time [13].

The process of standard model checking consists of three main tasks:

1 The first task is modelling, which is to convert a design into a formalism accepted by a model checking tool. In order to reduce the resources needed, the modelling of a design may require the use of abstraction to eliminate irrelevant details.

2 The second step is specification where it is necessary to state the properties that the design must satisfy. The specification is usually given in some logical formalism.

3 The third task is verification, which is ideally completely automatic. In practice, human assistance is usually needed to analyse the results.

The main challenge in model checking is dealing with the state space explosion problem. This problem occurs in systems with many components that can make transitions in parallel. During the past ten years considerable progress has been made in dealing with this problem [10, 20]. Much of the increase has been due to the use of binary decision diagrams (BDD), a data structure for representing Boolean functions.

3 Drawbacks of component-based development

3.1 Different environments

Component-based software development is ideally not more than putting pieces together. Traditional software development approaches make it possible for system developers to pre-check the compatibility of the individual parts of the system before the actual integration process takes place because in most cases the environment of the development and the deployment are the same [8, 21, 24]. On the other hand, CBSD uses an approach assuming that the original deployment environment of a component and its new deployment environment into which it is going to be integrated may be considerably different from each other. Even if the new environment is the same or similar, the component can be used in other, undiscovered ways, lacking any kind of validation concerning the original purpose it was meant to be used for.

However, CBSD would not be considered to revolutionize today's software development technologies if the use of prefabricated components could turn into such a major setback during the integration process compared to traditional techniques. Then what is the use of verifying the individual components if all that information on their correctness becomes irrelevant and out-of-date from the moment they are used anywhere but the original environment? The answer is based on the idea of building correct programs in which reliability is built in. This means the correctness properties of a program may be transferable or at least checkable in other environments as well. This concept led to the method of "design-by-contract" introduced by Bertrand Meyer in Object-Oriented Programming (OOP) [16]. According to him, defining a precondition and a postcondition for a routine is a way to define a contract that binds the routine and its callers. The notion of contracts is now extended to components where different reuse and deployment considerations become important, parameterizing the pre- and postconditions of components' contracts. This extension was proposed by Reussner and others and was referred to as parameterized contracts according to the "architecture-by-contract" principle [19].

The idea of using contracts in combination with built-in testing technologies according to Hans Gerhard Gross would offer a long-waited solution to increase our confidence in third-party components [9]. Built-in testing usually refers to all additional information for checking assertions and conditions at runtime with the exception of the assertions associated with the code of a component. Built-in testing is usually not part of the original functional requirements of a component and it does not stay incorporated in the code after the release of a component, but the author in [9] argues that components should be developed with permanent built-in testing capabilities from the beginning.

Built-in contract testing is based on the notion of providing

individual components with the ability to be checked by their execution environment at runtime. Whenever a component is deployed in a new environment, the built-in contract test part of it is going to check whether the component, the partner components and the environment like one another.

The meta information in contracts can be distinguished on which level and for what purposes they can be used. Beugnard distinguishes between four levels of increasingly negotiable properties [4]:

– Basic contracts (a syntactic contract) are specified either at the programming language level or through some IDL provided by the component platform.

– Behavioural contracts define the overall functionality of a component in terms of pre- and postconditions of operations and externally visible provided and required state transitions.

– Synchronization contracts add another dimension to the behavioural contract by providing the necessary information to coordinate the interdependent operations.

– Quantitative contracts or quality-of-service contracts quantify the expected behaviour or the component interaction in terms of minimum and maximum response delays, average response, quality of a result. This category would allow the comparison of different components built for the same purpose but with various component models by using measurable data.

Focusing on the interoperability of components the most significant is the type of behavioural contracts but from an educational point of view and in the concurrent programming world quantitative contracts can also play a key role.

3.2 Granularity of components

Defining the range of component granularity can also be difficult because several factors (level of abstraction, likelihood of change, complexity of a component, etc.) have to be considered while designing components. A component should not be too small as the interaction between smaller components requires more time and resources, on the other hand a component should not be too large as it provides more complex interfaces, is subject to more frequent changes and makes a system using it less flexible.

That is why it is essential to find a balance between the factors of cohesion and coupling. One of the main purposes of the educational framework outlined in [7] is to support to practice those design decisions. A similar framework (CDT) was developed by the authors of [3]. However, their framework is focusing primarily on the testability of third-party components outlining a possible solution using XML and does not contain any educational aspects.

4 An example: Automatic Betting Machine (ABM)

In our example we designed an automatic betting machine for the game “TippMix”, the Hungarian version of “Bet and Win”. The only input needed for the betting machine is the number of events a person wants to bet on (two-digit numbers entered on a console), and produces a ticket that is bound to the rules of the game with a combination of bets. A short overview of the game can be read below.

Customers of “TippMix” can bet on the outcome of events listed in the betting offers issued by “Szerencsejatek Zrt.” every week or every two weeks typically containing 200 betting events. (An optional input parameter may be the actual number of events for a given week.) Bets can be placed on the outcome of at least 1 and not more than 14 events in single bets or a combination of events in multi bets. In single bets the customer wins if they get all the outcomes of the events right they placed a bet on. In multi bets the customer can win with 1, 2 or 3 missed outcomes as well – depending on the type of the combination. The most significant constraints concerning the game are listed below:

– Only one bet can be placed on any one betting event.

– Bets can be placed on any combination of events.

– If you place bets on less than 3 events, these have to be selected from the first 50 events.

– If you place bets on 3 to 5 events these have to be selected from the first 150 events.

– If you place bets on 5 or more events these can be selected from all 200 events.

The simplified functioning of the automatic betting machine can be seen in Figure 1.

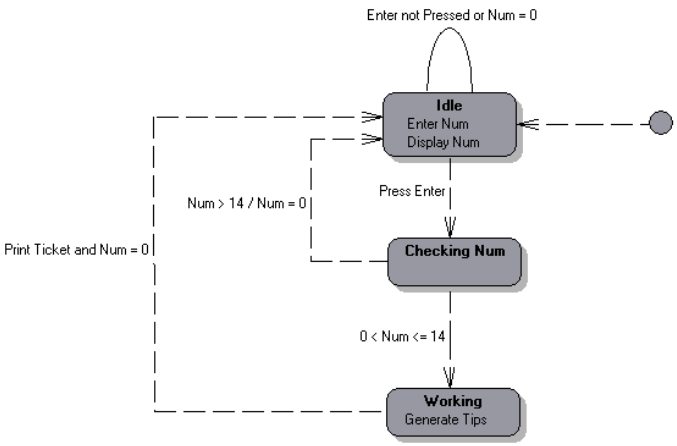

Fig. 1. A simplified state diagram of the ABM

There are different conditions that need to be satisfied in order to make a transition from one state to another. These conditions can guarantee the end results after leaving a state. These constraints are collected into the behavioural specification of the component, listed in Table 1.

Tab. 1. The behavioural specification of the ABM

Initial State and Precondition	Transition	Final State and Postcondition
Idle and Enter not Pressed	Waiting	Idle and Display(0)
Idle and Enter not Pressed	EnterNum(Num)	Idle and Display(Num)
Idle and EnterNum(0)	Press Enter	Idle and Display(0)
Idle and EnterNum(Num)	Press Enter	Checking Num and Display(Num)
Checking Num and $1 \leq Num < 3$		Working and Display(Num) and $Max = 50$)
Checking Num and $3 \leq Num < 5$		Working and Display(Num) and $Max = 150$)
Checking Num and $5 \leq Num \leq 14$		Working and Display(Num) and $Max = 200$)
Checking Num and $Num > 14$	Error	Idle and Display(0)
Working and $Num = 1$	Generate Tips and Print Ticket	Idle and $EventNum \leq 50$ and Tip in ("Home", "Draw", "Guest")
Working and $Num > 1$	Generate Tips and Print Ticket	Idle and $EventNum_i$ in $(1..Max)$ Where $i = 1..Num$ and $EventNum_i \neq EventNum_j$ Where $i \neq j$ and Tip in ("Home", "Draw", "Guest")

5 Verification of the system

5.1 The NuSMV Model Checker

SMV (Symbolic Model Verifier) is a tool for checking that finite-state systems satisfy specifications given in Computation Tree Logic (CTL). NuSMV is originated from the reengineering, reimplementation and extension of SMV that is robust and close to industrial systems standards. The analysis of specifications expressed in Linear Temporal Logic (LTL) is also possible in this version.

In NuSMV the specification of the system – usually a state transition machine – and the constraints imposed on its functioning expressed by CTL formulas are present together just as it can be seen in the following (oversimplified) excerpt of the main module.

```
MODULE main

VAR
state : {idle,checking,working} ;
num : {0..99} ;

ASSIGN
init(state) := idle ;
init(num) := 0 ;

next(state) :=
case
(state = idle) & (num > 0) : checking ;
(state = checking) & (num <= 14) : working ;
(state = working) : {working,idle} ;
1 : idle ;
esac ;

next(num) :=
case
(num < 99) : num + 1 ;
1 : 0 ;
esac ;

FAIRNESS !(state = checking)
FAIRNESS !(state = working)

SPEC AG((state = checking) -> AF(state = idle))
SPEC AG((state = checking) -> EF(state = working))
SPEC AG((state = working) -> AF(state = idle))
```

The variable *num* in the module is equivalent with the input data Num from the behavioural specification in Table 1. An oversimplified algorithm is used to simulate the input of *num*. Within a short time NuSMV returns the result that none of the constraints has been violated, which is quite easy to see.
Fig. 2 shows how the ABM is composed of two components and how they interact with each other. Labels ex_1, ex_2, ex_3 and re_1 are explained in Section 6.

5.2 Extending the sample system with multiple terminals

Now we consider a situation where the previously discussed ABM should be able to handle two terminals from which input data can arrive. We require mutual exclusion between the terminal components as only data from one terminal can be processed at a time. This is provided by a scheduler using a variant of the method introduced by Peterson [18]. The specification and the constraints are extended from the previous "base".

```
MODULE main

VAR
state0 : {idle,entering,checking,working} ;
state1 : {idle,entering,checking,working} ;
turn : boolean ;
term0 : process terminal(state0,state1,turn,0) ;
term1 : process terminal(state1,state0,turn,1) ;

ASSIGN
init(turn) := 0 ;
```

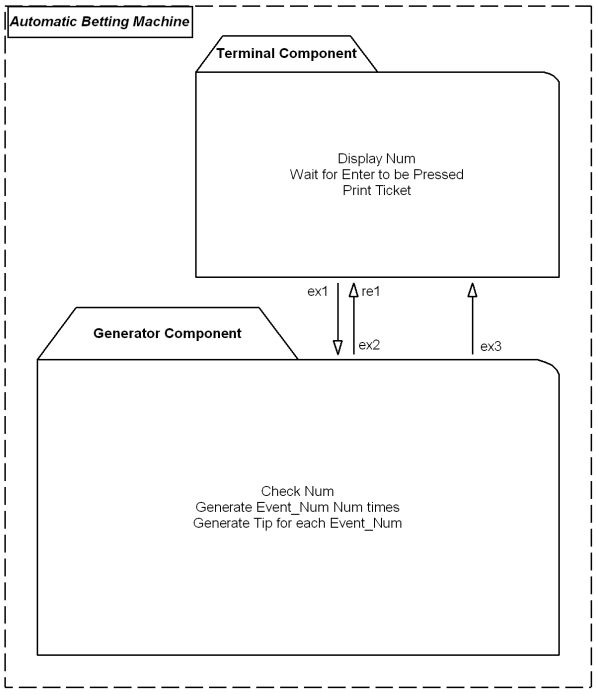

Fig. 2. Components of the ABM

```
FAIRNESS !(state0 = checking)
FAIRNESS !(state1 = checking)
FAIRNESS !(state0 = working)
FAIRNESS !(state1 = working)

SPEC AG !((state0 = checking) & (state1 = checking))
SPEC AG !((state0 = checking) & (state1 = working))
SPEC AG !((state1 = working) & (state0 = checking))
SPEC AG !((state0 = working) & (state1 = working))
SPEC AG((state0 = entering) -> AF(state0 = checking))
SPEC AG((state1 = entering) -> AF(state1 = checking))
SPEC AG((state0 = checking) -> AF(state0 = idle))
SPEC AG((state1 = checking) -> AF(state1 = idle))
SPEC AG((state0 = checking) -> EF(state0 = working))
SPEC AG((state1 = checking) -> EF(state1 = working))
SPEC AG((state0 = working) -> AF(state0 = idle))
SPEC AG((state1 = working) -> AF(state1 = idle))

MODULE terminal(state_sajat,state_masik,turn,turn_sajat)

VAR
num : {0..99} ;

ASSIGN
init(state_sajat) := idle ;
init(num) := 0 ;
next(state_sajat) :=
case
(state_sajat = idle) & (num > 0) : entering ;
(state_sajat = entering) & (state_masik = idle) : checking ;
(state_sajat = entering) & (state_masik = entering)
& (turn = turn_sajat) : checking ;
(state_sajat = checking) & (num <= 14)
& (turn = turn_sajat) : working ;
(state_sajat = checking) & (num > 14)
& (turn = turn_sajat) : idle ;
(state_sajat = working) & (turn = turn_sajat) : {working,idle} ;
1 : state_sajat ;
esac ;
next(num) :=
case
(num < 99) : num + 1 ;
1 : 0 ;
esac ;
next(turn) :=
case
(turn = turn_sajat) & ((state_sajat = checking) |
 (state_sajat = working)) : !turn ;
1 : turn ;
esac ;

FAIRNESS
running
```

Here the constraints are not violated, either. However, strictly for educational purposes some of the constraints can be changed because the counterexample generated by NuSMV can help the understanding of concurrent programming, for instance.

The next question is whether it is always necessary to check the entire system model or it is sufficient if the constraints satisfied in the base component are matched against the constraints in the extension.

6 Open Incremental Model Checking (OIMC)

OIMC introduced in [23] addresses the changes to a system instead of re-checking the entire system model including the new extensions. The model checking is executed in an incremental manner within the extension component only. A simplified model of the ABM extended from the previous model in such a way can be seen in Fig. 3.

The conditions – basically inter-component constraints – resemble to pre- and postconditions in "design-by-contract". The definition they used to describe a software system is given bellow:

A state transition model M is represented by a tuple $\langle S, \Sigma, s_0, R, L\rangle$ where S is a set of states, Σ is the set of input events, $s_0 \in S$ is the initial state, $R \subseteq S \times PL(\Sigma) \rightarrow S$ is the transition function (where $PL(\Sigma)$ denotes the set of guarded events in Σ whose conditions are propositional logic expressions), and $L : S \rightarrow 2^{AP}$ labels each state with the set of atomic propositions true in that state.

Components can be verified to be consistent via OIMC. Initially, a CTL property p is known to hold in B where B is the base component (the original betting machine in our example). We need to check that E (standing for the extension component) does not violate p. The incremental verification method only needs to verify the conformance at all exit states between B and E. Corresponding to each exit state ex, within E, the algorithm to verify preservation constraints $v_B(ex, cl(p))$ can be briefly described as follows [23]:

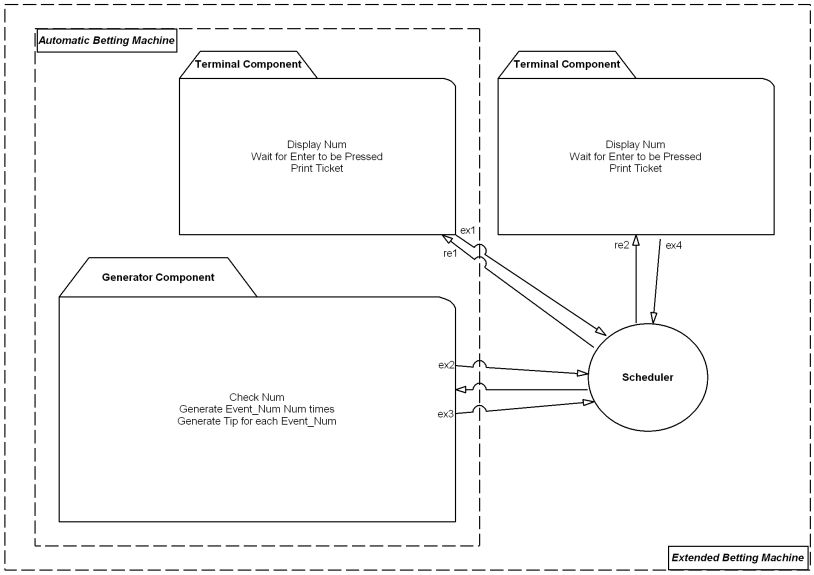

Fig. 3. The model of the extended ABM

1 Seeding $v_B(re, cl(p))$ at any reentry state re where $cl(p)$ is the set of all sub-formulae of p including itself. The assumption function As is: $As(re) = v_B(re, cl(p))$.

2 Executing a CTL assumption model checking procedure in E to check ϕ, $\forall\phi \in cl(p)$. In case of COTS, a standard CTL model checking is executed within E instead.

3 Checking if $v_E(ex, cl(p)) = v_B(ex, cl(p))$.

At the end of the algorithm, if at all exit states, the truth values with respect to $cl(p)$ are matched respectively, B and E are consistent with respect to p.

In our example the original betting machine consists of two components interacting with each other. The terminal components have one exit state (ex_1) when Enter is pressed and one reentry state (re_1) when the two-digit number entered is greater than 14 and the terminal returns to the initial state. The generator component has one exit state (ex_2) when a ticket is returned to be printed at the terminal. The extended betting machine consists of two terminal components, the generator and a scheduler which we will not consider as it has no effect on the functionality of the machine. There is no direct connection between any of the terminal components and the generator anymore. The two constraints below (and the closure for each) can easily be checked at the exit state ex_1.

```
SPEC AG((state = checking) -> AF(state = idle))
SPEC AG((state = working) -> AF(state = idle))
```

In the next step the extension must be checked with respect to the properties described above and their closure sets. The same properties appear twice in the list below as there are two terminal components in the extended system.

```
SPEC AG((state0 = checking) -> AF(state0 = idle))
SPEC AG((state1 = checking) -> AF(state1 = idle))
SPEC AG((state0 = working) -> AF(state0 = idle))
SPEC AG((state1 = working) -> AF(state1 = idle))
```

The properties in connection with the scheduler component are not considered as they do not affect the interoperability between the terminals and the generator. These are listed here.

```
SPEC AG !((state0 = checking) & (state1 = checking))
SPEC AG !((state0 = checking) & (state1 = working))
SPEC AG !((state1 = working) & (state0 = checking))
SPEC AG !((state0 = working) & (state1 = working))
SPEC AG((state0 = entering) -> AF(state0 = checking))
SPEC AG((state1 = entering) -> AF(state1 = checking))
```

There are two properties present in the original betting machine that do not need to be checked again as they refer to a state transition within the generator component, not at any exit or reentry states. These are the following:

```
SPEC AG((state0 = checking) -> EF(state0 = working))
SPEC AG((state1 = checking) -> EF(state1 = working))
```

Finally, the algorithm compares whether the truth values of the closure sets of the properties at the exit states are equivalent or not. In the case of the ABM it is relatively easy to see that the algorithm ends with a successful comparison of the truth values. In more complex systems the number of properties that do not need to be re-checked is expected to increase.

The theorem concerning with the scalability of the OIMC method has already been proved in [17] and can be read below:

Theorem 1 *If all respective pairs of base* (C_{i-1}) *and refining* (E_i) *components conform, the complexity of OIMC to verify the consistency between* E_n *and* B *is independent from the* n*-th version of* C_n*, i.e. it only executes within* E_n*.*

Despite the proof further studies are necessary to see the efficiency of OIMC in real applications using third-party components-off-the-shelf. Also, it is quite important to integrate the method of OIMC into a commonly used model checker tool such as the previously mentioned NuSMV.

7 Future work

Circular dependency between interface states of the base and the extension cannot be handled by OIMC.

Although defining contracts for the component is still our job, there are remarkable results in this field as the authors of [1] created a tool that is capable of extracting contracts from components automatically, but only in the .NET environment. A similar tool for the Java platform would not only be appreciated by component developers and assemblers but it would also provide an opportunity to compare the contracts extracted by different tools. A tool should be developed which is able to transform these contracts to CTL temporal constraints for model checking (especially OIMC).

An educational framework with a library of verified components outlined in [7] needs to be created for providing an environment for IT students to practice design decisions.

8 Conclusions

We must not forget about the educational aspect of contracts either, as the effectiveness of verified components developed for the same purpose but by using different tools also becomes comparable based on measurable data (execution time, etc). The example of the automatic betting machine is also suitable to illustrate the design decisions concerning the granularity of components. If we assume that our client needs an automatic gambling machine supporting not only betting on sports events but also different lottery games, etc. then the developer has to decide which case is the most suitable for the specific task.

– The first alternative is to develop the remaining components from scratch not paying the least attention to the already existing component.

– The second solution is to try to adapt the existing component to the modified situation. The developer may recognize the similarity between betting on sports events and a lottery game with five numbers in the following way. A lottery number can be considered a sports event, the number of all sports events is 90 and the number of bets placed on sports events is always five. The bet itself is not important in this scenario.

– The third alternative is to restructure the entire component, for example the random number generator becomes an independent component as it is the most reusable part of the software in other gambling games as well.

The best solution depends on the specific situation but it is notable that it may be another aspect to compare component efficiency for educational purposes. Hopefully this research can result in the better use of formal methods in practical applications, the better understanding of specifications, and finally less debugging work for developers. The ultimate goal is to make the processes of integration testing and checking the entire system model unnecessary for large and complex systems in the case of replaced or modified components. As a consequence of that third-party components formally verified by model checking techniques become more reliable and trustable in practice.

References

1 **Arnout K, Meyer B**, *Finding Implicit Contracts in .NET Components*, Proceedings of FMCO 2002 (Formal Methods for Components and Objects), Springer-Verlag, Leiden The Netherlands, August 5. LNCS 2852, 2002.

2 **Bass L, Clements P, Kazmar R**, *Software Architecture in Practice (Second Edition)*, Addison-Wesley, 2003.

3 **Bertolino A, Polini A**, *A Framework for Component Deployment Testing*, Proceedings of the 25th International Conference on Software Engineering, 2003.

4 **Beugnard A, Jézéquel JM, Plouzeau N, Watkins D**, *Making components contract aware*, IEEE Software **32** (1999), 38-44. Issue 7.

5 **Clarke E, Grumberg O, Peled D**, *Model Checking*, MIT Press, 2000.

6 **Crnkovic I, Hnich B, Jonsson T, Kiziltan Z**, *Specification, Implementation and Deployment of Components*, Communications of the ACM **45** (2002), 35-40.

7 **Dávid Á, Pozsgai T, Kozma L**, *Educational framework for developing applications built from verified components*, Proceedings of the Ninth Symposium on Programming Languages and Software Tools, Tartu University Press, Tartu Estonia, Spring Aug 13, 2005.

8 **Fayad ME, Hamu DS, Brugali D**, *Enterprise frameworks characteristics, criteria and challenges*, Communications of the ACM **43** (2000), 39-46.

9 **Gross HG**, *Component-Based Software Testing with UML*, Springer, 2005.

10 **Hatcliff J, Deng W, Dwyer M, Jung G, Prasad V**, *Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems*, Proceedings of the 25th International Conference on Software Engineering, 2003, pp. 160-173.

11 **Hopkins J**, *Component primer*, Communications of the ACM **43** (2000), 27-30.

12 **Kobryn C**, *Modeling components and frameworks with UML*, Communications of the ACM **43** (2000), 31-38.

13 **Kroger F**, *Temporal Logic of Programs*, Springer-Verlag, 1987.

14 **Larsen G**, *Component-based enterprise frameworks*, Communications of the ACM **43** (2000), 25-26.

15 **Lee SC, Shirani AI**, *A component based methodology for web application development*, Journal of Systems and Software **71** (2004), 177-187. Issue 1-2.

16 **Meyer B**, *Object-Oriented Software Construction (Second Edition)*, Prentice Hall, 1997.

17 **Nguyen TT., Katayama T**, *Handling consistency of software evolution in an efficient way*, Proceedings of the IWPSE, 2004, pp. 121-130.

18 **Peterson GL**, *A new Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables*, ACM TOPLAS **5** (1983), 56-65.

19 **Reussner RH**, *The use of parameterised contracts for architecting systems with software components*, 6th Intl. Workshop on Component-Oriented Programming, 2001.

20 **Robby A, Dwyer MB, Hatcliff J**, *Bogor: An Extensible and Highly-Modular Software Model Checking Framework ACM SIGSOFT Software Engineering Notes*, Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, 2003.

21 **Sparling M**, *Lessons learned through six years of component-based development*, Communications of the ACM **43** (2000), 47-53.

22 **Szyperski C**, *Component Software Beyond Object-Oriented Programming (Second Edition)*, Addison-Wesley/ACM Press, 2002.

23 **Thang NT, Katayama T**, *Specification and verification of inter-component constraints in CTL*, Proceedings of the 2005 conference on Specification and verification of component-based systems, Vol. 31, ACM Press. Issue 2, 2005.

24 **Vitharana P**, *Risks and Challenges of Component-based Software Development*, Communications of the ACM **46** (2003), 67-72.