Annales Univ. Sci. Budapest., Sect. Comp. 36 (2012) 323–339

DEFINING CONTRACTS WITH DIFFERENT TOOLS IN SOFTWARE DEVELOPMENT

György Orbán and László Kozma
(Budapest, Hungary)

Communicated by Zoltán Horváth

(Received December 22, 2011; accepted January 22, 2012)

Abstract. To create reliable software systems, different tools and methods are needed. To analyse the available tools and methods, two main software development levels were chosen. The first is the model (UML with OCL, ADL, JML) level. This is important because with models different checks can be made, which can help to find software failures earlier. However, the checks at the model level are not enough. Analysis was also made on the second, implementation level (Eiffel, Java, .NET), where the programming language extensions for contract based development were examined.

1. Introduction

Reliability is very important in component based development. To create verified components which can be used to build software systems, it is necessary to use different tools and methods which support the creation of better quality software systems. These tools and methods should support different parts of the development processes. Software development processes should start with

Key words and phrases: Contracts, component based software development, reliability of software.
1998 CR Categories and Descriptors: D.2.4.
The Research is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).

analysis, design and modelling. Creating the models of the developed system can have many advantages. With models many design errors can be found before the actual implementation.

One of the available methods that support the creation of better software is contract based development. To create software systems with contracts, there should be tools at the design levels and at the implementation levels too.

In this paper we will focus on the role of contracts in software development processes at the modelling level and at the implementation level as well.

We analyse how the available and continuously developed tools support the creation of the high quality software systems. A simple bank example was chosen to introduce how the contract based development methodologies can be used in system models and in the implementations. A bank can have many accounts, and we created the contracts to define the behaviour of the actions on the accounts. We can deposit some money and we can withdraw some money. To define the proper behaviour of these methods (deposit, withdraw), different contracts were defined.

After introducing the opportunities by the different tools, we examine how the defined contracts are checked at runtime.

2. Contract based development

The support for contract based development is a continuously developing area. The "Design by contract" term was created by Bertrand Meyer [24]. He developed a programming language (Eiffel [14, 9]), which supports the "Design by contract" paradigm. Now there are many programming languages (Eiffel, D [5], etc.) with native support, or (Java[12], C++, .NET [22], Python[31]) an extension makes it possible to use contracts in the development processes.

With these tools more reliable software systems can be created with better quality. For this purpose contracts can be used which has three main types (preconditions, postconditions, invariants). Using contracts benefits and obligations can be described for the client and for the provider too. With a precondition we define the conditions needed before the method call or component usage, to work properly. Postconditions define obligations for the provider and benefits to the client. They make sure if preconditions are satisfied by the client; the provider will work as it is defined in the specification and so in the contracts. Invariants contain requirements which should be true all the time.

Using contracts it is possible to define the requirements at the implementation and at the design levels. These definitions are not separated from the model or the implementation. This can help software designers and developers in their work. The ability of adaptation to the specification can be increased by updating the contracts. After the update (contract change) it will be clear that the system still can work with the new specifications, or other changes in the implementation are needed.

The usage of contracts can have more benefits. For example: if the requirements are documented in the source code, it is easier to understand the implementation for the maintainer or it is easier for the developer to make changes to the implementation. Maintenance and software upgrade costs can be more than the actual development costs, but with the usage of contracts the costs can be decreased.

3. Component based development

The software development processes have changed. Nowadays the software systems are not built from scratch, but reusable elements are the building blocks of the new systems. This is possible with component based development and the software system can be built faster. If the system needs some new functionality, the necessary component has to be found and connected to the system. However, this connection is not always easy. Sometimes some extra modules or glue code have to be developed to connect the different parts together. To make the connection of the components as easy as possible a proper formal description of the component is needed. There is a need for this because the component is often only a black box for the component user.

In the connection process the most important part of the component is the interface. There are two main types of component interfaces: the provided and the required interfaces. On the provided interface the component provides its services to other clients, and on the required interfaces it uses services from other parts of the system (component or environment) to work properly.

The extension of the interfaces with contracts can make the components more reliable and has more advantages if the system is built from separate developed components. If we extend the component interface with preconditions, the interface can check if the connection and communication between the components is possible. With the postconditions and invariants the components reliability can be increased.

4. Support for contracts in software system models

Every software development process should be started with analysing the requirements and design the software system. This can be made with the usage of more formal requirement specification languages (ADL, RSL [37], Z [36], VDM [26], etc.) or with less formal or semi-formal modelling languages (UML [29], etc.). For every domain the most appropriate method should be chosen.

In this section we focus on how the contract based development can be supported in the software system models. This can be made before the actual implementation of the software system, and if there is a support to verify the models, many design errors are avoidable. This can save money and time when the concrete implementation is made.

The three examined model level technologies are UML models extended with OCL [28] descriptions, architecture description languages and Java modeling language. All of these technologies support contracts in different ways.

4.1. UML and OCL

UML (Unified Modeling Language [29]) represent a semi formal system description method. It has many benefits, because it is more visual (and so easier to understand) than a formal description with mathematical formulas in it, but that is why it is much harder to verify a model created in UML. When the UML model needs to be extended with constraints, OCL is a good choice. OCL is a formal language for software developers and designers: it is a specification language to extend the UML models. The statements written in OCL can be evaluated, but this has no effect on the model. OCL has many purposes like describe preconditions and postconditions on methods, specify invariants on classes, describe guards, specify constraints on operations. Design by contract methods can be used in the UML models with the support of OCL (Listing 1).

```
context Account
inv: self.accountbalance >= 0

context Account:deposit(money)
pre: money > 0
post: self.accountbalance = self.accountbalance@pre + money

context Account:withdraw(money)
pre: money > 0 and self.accountbalance > money
post: self.accountbalance = self.accountbalance@pre − money
```

Listing 1. OCL contract definition

The extension of the models are only at the design level, but contracts are needed at the implementation level too. This could be done with a source code generator like DresdenOCL [7]. DresdenOCL supports Java source code generation from UML models, from EMF Ecore-Based Models and from the imported Java classes with the OCL extension.

In this simple example we define invariants, preconditions and postconditions for our bank account example. So assume that we have an Account class and two methods to deposit and withdraw to modify the balance.

4.2. Architecture Description Languages

Architecture Description Languages (ACME [1], Rapide [32], Wright [38], Unicon [39] etc.) represent a higher level architecture description. The ISO/IEC 42010 [11] defines that ADL specifications are a more formal description of the system. These descriptions focus on the components and the connections (interfaces) between them in the software system. But there are many ADL languages like Rapide with the support for an abstract description of the behaviour of the software system.

A similar approach is developed in the SOFA [34] component system. It supports a hierarchical component model and features as dynamic architectures, multiple communication styles, composition and behaviour verification etc. SOFA supports the ADL based component development. In this case, the ADL is an XML file describing single top level entity. To describe the components behaviour, SOFA uses a Behaviour Protocol [18]. This behaviour protocol can be used as contracts if we look at the required interfaces as a precondition and the provided interface as a postcondition.

In this case we can talk about parametrised contract [33], which is a mapping between the required interfaces and the provided interfaces. The behaviour of each interface (and so the component) is described by protocols. These protocols are modelled as a finite state machine. These finite state machines describe the behaviour of the component, which are a set of the method call sequences.

With these formal descriptions (the architecture description and the behaviour description) and the developed tools (dChecker [34], BP2Promela [34], BPTools [34]) the verification of the components is possible. With the dChecker communication errors between the components can be detected. This makes the component integration easier. The BP2Promela tool can transform Extended Behavior Protocol (EBP) models to Promela language, which is the verification modeling language for the SPIN [35] model checker. With this conversion further verification of the component system can be made in the SPIN model checker.

4.3. JML

JML (Java Modeling Language) [15, 16] is a formal behavioural interface specification language. It allows specifying syntactic interface of Java source code and its behaviour too. This technology is important when developing software components in Java environment. JML (Listing 2) supports the contract based development [19] in a similar way to the implementation level, where preconditions, postconditions and invariants can be defined to describe the behaviour of the system.

Contracts written with JML need a special annotation. This annotation is similar to source code comments and the contracts can be written before the method implementation in the source code.

```
//@ requires money>=0;
//@ ensures (balance == \old(balance) + money);
public static deposit(int money)

//@ requires money>=0;
//@ ensures (balance == \old(balance) - money);
public static withdraw(int money)
```

Listing 2. JML contract definition

Java expressions can be used to define the contracts. JML is developed so that it can be used to support many different tools which support runtime assertion checking, invariant detectors (Daikon [6]), or tools which support creation, analysis and verification of object oriented software systems (KeY [17], etc.).

There is a support in the JML descriptions for informal specifications. These can be useful several times, for example when there is no time to create a formal description. There is no support for automatic processing (assertion checking) of the informal contracts, but they can be still useful.

There are some keywords, expressions which can be used in JML descriptions. Some of them is similar to the keywords used by the tools at the implementation level. The similar keywords are \old, \result; these mean the same things [16]. There are some restrictions for the expressions which can be created in JML. These expressions can not have any side effects, and only pure methods (which do not change an objects state) can be used in assertions.

Information hiding is also supported, which is necessary if interfaces are specified. The interface specification cannot contain any private data, because the client cannot depend on it. If the private data are hidden from the interface user, the component behind the interface can be changed without any consequences.

In JML expression quantifiers can be used. With these quantifiers: universal quantifier(\forall), existential quantifier(\exists), generalized quantifiers (\sum, \product, \min, \max) and numeric quantifier (\num_of) better and more meaningful expressions, predicates can be formalized, so the behaviours can be described more precisely.

JML represents a higher level according to the implementation level, but there is a strong connection between them. One of the things that represents this higher level is the model variables (specification-only variables) in the expressions. In the short example (Listing 3) a model variable can be seen. In this case, there is a mapping between the abstract model (name) and the more concrete variable (accountOwnerName). AccountOwnerName is private hidden from the client. These are very useful when the implementation should be changed with the usage of a new data structure. In this case, there is no need to change the public specifications, which could affect the client code which depends on the specification.

```
//@public model non_null String name;
private /*@ non_null @*/ String accountOwnerName;
//@private represents name <- accountOwnerName;
```

Listing 3. JML model variable

JML does not only support model variables: it also supports model methods, classes and interfaces.

The analysis of the presented modelling tools and methods can be found after the runtime contract checking techniques and methods chapter, because we would like to introduce a technological workflow built on the tools and methods starting from the system modelling to the actual implementation.

5. Support for contracts in the software system implementations

There are many options to use contracts in the system models. Many times the source code generation from these models is not an easy task. For the support of contracts in the actual implementation, programming languages (Eiffel, D etc.) with native contract based design support or programming languages (Java, .NET, C++ etc.) with continuously developed extensions can be used. This Section will focus on three programming languages: Eiffel, Microsoft .NET [22] and Java.

Eiffel has a native support for the "Design by Contract" paradigm, but the other two languages need to use extensions to support contracts.

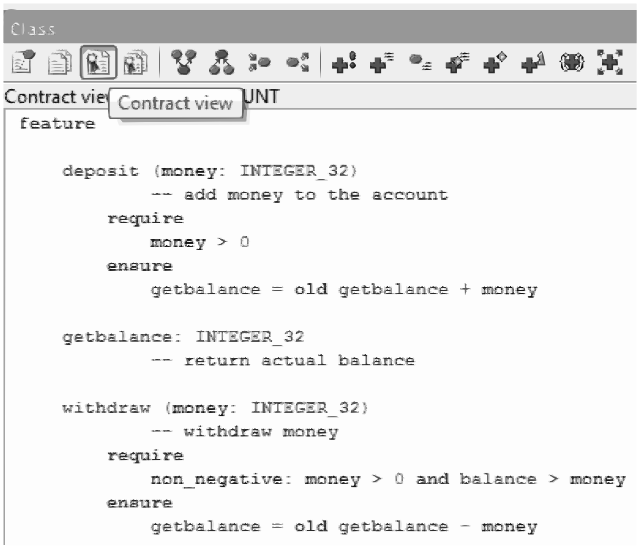
5.1. Eiffel

Eiffel is an object oriented programming language designed by Bertrand Meyer. The language is designed with some basic principles like ”design by contract”, which is a trademark of Eiffel Software. Eiffel is not only a programming language, it is a software development methodology too. Tools like EiffelStudio support these methods and technologies. It supports many software system views (models, source code Listing 4, etc.), which make the software development process easier.

```
class
        ACCOUNT
create
        make
feature --Initialize
        balance:INTEGER
                --balance

        make(money: INTEGER)
                        -- initialize account
                do
                        deposit(money)
                end
feature
        deposit(money : INTEGER)
                        -- add money to the account
                require
                        money > 0
                do
                        balance := balance + money
                        print (”Desposit!%N”)
                ensure
                        getbalance = old getbalance + money
                end
feature
        withdraw(money : INTEGER)
                        -- withdraw money
                require
                        non_negative: money > 0 and balance > money
                do
                        balance := balance - money
                        print (”Withdraw!%N”)
                ensure
                        getbalance = old getbalance - money
                end
invariant
        balance >= 0

end
```

Listing 4. Eiffel source code

Changing the views is easy when it is needed; source code can be edited to create the actual implementation, or when further design is needed the model view can be used. There is another useful view, which is contract view. With contract view (Figure 1) only the contracts related to the methods and classes can be seen.



```
feature

    deposit (money: INTEGER_32)
            -- add money to the account
        require
            money > 0
        ensure
            getbalance = old getbalance + money

    getbalance: INTEGER_32
            -- return actual balance

    withdraw (money: INTEGER_32)
            -- withdraw money
        require
            non_negative: money > 0 and balance > money
        ensure
            getbalance = old getbalance - money
```

Figure 1. Eiffel contract view

This can be used by the client developers to see what the requirements (preconditions) are to access some feature and what that feature guarantees (postconditions). In this paper we just introduced some of the many tools in EiffelStudio which support the design by contract principle. These tools make the contract based development processes more easier compared to other tools.

5.2. Microsoft .NET

A Microsoft research project is Code Contracts [20]. The aim of this research project is to develop a Microsoft Visual Studio extension which supports the Design by Contract methods in .NET environment (Listing 5).

This extension has three main parts (runtime checking, static checking and document generation).

With the support for contract based development preconditions, postconditions and invariants can be used during the development of .NET software as the example shows (Listing 5).

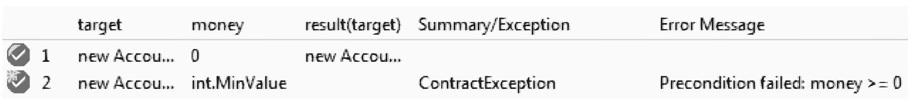
There is another extension to Microsoft Visual Studio, which is PEX [23], an automated parameterized unit test generation tool. The great advantage of this tool is that it can be connected with Code Contracts to generate more meaningful unit tests.

In this short .NET source code contracts are implemented in the deposit and in the withdraw methods. This contracts can be checked at compile and at runtime too. If there is some contract violation, an exception is thrown.

```
class Account
{
 int balance;

 public Account(int startBalance){
  balance = startBalance;
 }

 public void deposit(int money){
 Contract.Requires(money >= 0);
 Contract.Ensures((Contract.OldValue(balance) + money) == balance);

  balance = balance + money;
 }

 public void withdraw(int money){
 Contract.Requires(money >= 0);
 Contract.Requires(money <= this.GetBalance());
 Contract.Ensures((Contract.OldValue(balance) - money) == balance);

  balance = balance - money;
 }
}
```

Listing 5. .NET Code Contracts

There are preconditions (Requires) and postconditions (Ensures) for the deposit and the withdraw methods. This contract can help the testing processes together with the Pex tool. With the Pex Visual Studio extension and the source code contracts it is possible to automatically create more meaningful parameterized unit tests for .NET applications (Figure 2).



	target	money	result(target)	Summary/Exception	Error Message
1	new Accou...	0	new Accou...		
2	new Accou...	int.MinValue		ContractException	Precondition failed: money >= 0

Figure 2. PEX result

As we can see (Figure 2), Pex has automatically generated several unit tests for the methods with different parameters, and if there is a contract violation, an exception is thrown. There are different parameters generated based on the given contracts. In the first test case the input value is 0, which is a valid number. It can be accepted based on the given contracts, but in the second test case it tries the minimal value of an integer, which is a negative number. In this case the preconditions were violated, so an exception was thrown.

5.3. Java

Java does not have native support for the contract based design, so there is a need to use extensions if the developer wants to have the benefits of contract based design. Some projects (Contracts4J [4], jContractor [13], Modern Jass [25], etc.) tried to add the contract support to the Java language, which is a big challenge. Many projects are not developed any more, but there are some continuously developed extensions like Contracts for Java [3]. With the contracts for Java it is possible to use preconditions, postconditions and class invariants in the Java source code. Contracts for Java supports runtime contract checking, so when a contract is violated a java exception is thrown.

```
class Account
{
 int balance;

 Account(int StartBalance){
  balance = startBalance;
 }

 @Requires({"money > 0"})
 @Ensures({balance = old(balance) + money})
 void deposit(int money){
   balance = balance + money;
 }

 @Requires({"money > 0"})
 @Requires({"money < getBalance()"})
 @Ensures({balance = old(balance) - money})
 void withdraw(int money){
   balance = balance - money;
 }

 int getBalance(){
   return balance;
 }
}
```

Listing 6. Java contracts

Compared to Code Contracts (.NET) it has no support for static checking or to generate documentation, but it is an open source tool to support contract based design in the Java programming language. The sort example (Listing 6) will show how contracts can be used in Java. In our short Java source code we implemented some basic methods for an Account. With these simple methods the balance belonging to the Account can be modified. We have, for example, preconditions connected to deposit and withdraw methods, which define some obligations for the client who wants to modify the amount of the money. With the preconditions we define that only more than zero money can be added to the account.

6. Runtime contract checking techniques and methods

This chapter will analyse how contracts are checked at runtime in different programming languages (Eiffel, .NET and Java). Runtime contract checking is made with techniques supported by programming languages like assertions and exceptions, but there are many interesting questions like contract inheritance which need further research.

In Eiffel, there is a native support for contracts, and there is no need for contract weaving. Eiffel has its own compiler, which can compile the source code with contracts. In the compiled source code the contracts are checked at runtime. If there is a contract violation, runtime assertions and exceptions are used. In the assertions it is possible to define Boolean expressions too. These expressions may include function calls. With this function calls more meaningful contracts can be defined [8].

In .NET Code Contracts environment ccrewrite is the tool, which generates runtime checks from the defined contracts. It puts the runtime check at the appropriate places. Every contract usage is translated to an appropriate rewriter method [21]. It is possible to set in the rewriter that only an exception with or without an assertion should be thrown when a contract fails. The runtime contract methods can be generated with the rewriter or it is possible for the developer to create his own methods.

In Contracts for Java contracts are compiled separately not like in JML, where there is a compiler which replaces the Java compiler. The weaving has two types: online or offline, and this is made through bytecode rewriting.

At compile time the annotation processor is responsible to compile the contracts into a separate Java bytecode. When the classes are loaded, a Java agent rewrites the bytecode and weaves the contracts into the target methods. This weaving can be made separate of the Java launcher. In this case, the contracts are checked and always there is no need for a Java agent. There are two properties why contracts are compiled separately. The first is that "the compilation of contracts does not interfere with the normal compilation process", and the second is "the contract compiler is not needed for code with contracts to compile" [27]. To use this approach, compilation relies on the Java compiler and it is necessary that the Java compiler does not optimize across method boundaries. This is needed for the successful weaving of contracts bytecode with the contract free bytecode.

The contract compilation is made in two steps: a preprocessing step and an annotation processing step. After the compilation the contract files implement the full contract logic (inheritance too), so only the weaving is needed for the contract free bytecode to the proper call sites. The contracts are compiled into a "helper contract method" [27]. With this method the contracts will be evaluated. It has no knowledge of the context or about any inheritance issues. For every method with contracts a "contract method" will be created, which calls all the "helper contract methods" related to the method.

Interface contracts are compiled differently. Because interfaces cannot contain any code, all of the contracts are put in a helper class. Every interface with contracts has a helper class.

As we see runtime contract check is made with this helper method. In this methods the preconditions are combined with OR operator, and the postconditions, invariants are combined with the AND operator [3]. When a contract method fails, it throws an exception.

7. Analysis of the tools from the point of view of workflows

At the model level we examined ADL, UML with OCL and JML. ADL specification techniques are used in the SOFA2 component framework. In SOFA2 the system specification is made with ADL. To describe the behaviour of the components, Extended Behaviour Protocol (EBP) can be used. This behaviour protocol can be verified by the SPIN model checker. This verification is made at design level, where the system models are made. In the SOFA2 component framework the components are built in Java programming language at the implementation level. The communication between the components can be verified by SPIN model checker with the usage of the converted EBP to promela language. The conversion can be made using the BP2Promela tool [34]. The component behaviour can be defined with contracts using the contracts for Java tool. It would be useful to connect these two techniques, because the SOFA2 framework supports a higher level component system behaviour verification, and tools like contracts for Java support the software behaviour description at the implementation level.

Further research is needed to examine the possibilities of connecting the design level methods and tools (SOFA2) with the implementation level tools and techniques like Contracts for Java. In the other two model level methods, UML with OCL and JML, it is possible to generate source code from the contracts. For the Java source code generation from UML and OCL the DresdenOCL tool could be used, and for the JML there is a compiler, for example OpenJML [30] which is built on OpenJDK, which is an open source implementation of the javac compiler.

To put the created contracts into the actual implementation, different techniques provided by Java can be used. The defined constraints can be put in the compiled source code in different ways. This can be made with handcrafted constraints, code instrumentation (source code or bytecode), compiler based, explicit constraint classes, interceptor mechanisms [10].

When the models are created with UML and OCL, the contracts will be defined with OCL; these will be generated with wrapper based source code instrumentation into the source code by the DresdenOCL tool. Wrapper based constraint validation means that there will be a wrapper method generated which contains the contracts, and there will be a method call to the original method in the wrapper method. In general, the original method will be renamed [10].

In the case of using JML, a different method is used to generate source code. It has its own compiler built on an existing Java compiler, which is extended to understand the defined contracts, so Java bytecode can be generated from the JML model in one step. It is made with the usage of custom annotations. The drawback of the JML method is that a customized compiler is needed every time, in contrast with OCL based generation technique.

8. Conclusions

There are several formal verification possibilities during the creation of the system models. Many model checking tools (SPIN [35], CADP [2]) support the verification of the models with formal methods (XTL (eXecutable Temporal Language), etc.). At the implementation level the contracts make it possible to check the behaviour of the system at runtime.

We saw that the different tools for Java, Eiffel and for .NET support different methods to add the contracts to the running source code. UML extended with OCL and then converted with DresdenOCL uses wrapper based techniques to generate contracts into the source code. JML has a separate compiler to compile the contracts. Contracts for Java uses weaving and rewrites Java bytecode. .NET also uses a rewriter to put the contracts into the source code. Eiffel has native support and it can compile the contracts.

There is a need for the investigation of the advantages of using the "design by contract" approach in software system modelling and in the actual implementations. The introduced methods and tools can support the contract based development in different ways. If a higher level system verification is needed ADL languages like in SOFA2 component framework could be used. The model and the implementation level tools and techniques are very different and there is a need to connect them. As we suggested in Section 7, we can achieve this aim with using different kinds of tools, connecting them in different kinds of workflows.

In many software development projects there is no time for formal verification. A proper verification of a software system can take a lot of time and in a fast changing IT industry it can be crucial how fast an application, a new product can hit the market.

The fast application development should not affect the quality of the software product, and that is why new tools and methods are needed which support the creation of more reliable software systems.

References

[1] **Acme** - the acme architectural description language and design environment, `http://www.cs.cmu.edu/~acme/`, 2011.

[2] **CADP,** `http://www.inrialpes.fr/vasy/cadp/`, 2011.

[3] **Contracts for Java** (cofoja), `http://code.google.com/p/cofoja/`, 2011.

[4] **Contract4J**, `http://polyglotprogramming.com/contract4j`, 2011.

[5] **D programming language,** `http://www.d-programming-language.org/index.html`, 2011.

[6] **Daikon,** `http://groups.csail.mit.edu/pag/daikon/`, 2011.

[7] **Dresden OCL,** `http://www.dresden-ocl.org/index.php/DresdenOCL`, 2011.

[8] **Eiffel online documentation,** `http://docs.eiffel.com/`, 2011.

[9] **Eiffel software,** `http://www.eiffel.com/`, 2011.

[10] **Lorenz, F., G. Glos, J. Osrael, and M.K. Goeschka,** Overview and evaluation of constraint validation approaches in Java, in: *29th International Conference on Software Engineering (ICSE'07)*, 2007, 313–322.

[11] **ISO/IEC/IEEE 42010,** `http://www.iso-architecture.org/42010/`, 2011.

[12] **Java,** `http://www.java.com/`, 2011.

[13] **jContractor,** `http://jcontractor.sourceforge.net/`, 2011.

[14] **Jézéquel, J.-M.,** *Object-oriented Software Engineering with Eiffel*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[15] **JML - Java Modeling Language,** `http://www.cs.ucf.edu/~leavens/JML/`, 2011.

[16] **JML - Java Modeling Language,** *Reference manual*, `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/`, 2011.

[17] **KeY,** `http://www.key-project.org/`, 2011.

[18] **Kofron, J.,** *Behavior Protocols Exensions*, Doctoral Thesis, Charles University in Prague, `http://d3s.mff.cuni.cz/~kofron/phd-thesis/`, 2011.

[19] **Leavens, T.G. and Y. Cheon,** *Design by Contract with JML*, `http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf`, 2006.

[20] **Microsoft Code Contracts** - microsoft research, `http://research.microsoft.com/en-us/projects/contracts/`, 2011.

[21] **Microsoft Corporation,** *Code Contracts User Manual*, 2011

[22] **Microsoft Corporation,** Microsoft .NET framework, `http://www.microsoft.com/net/`, 2011.

[23] **Microsoft Pex,** automated white box testing for .NET - microsoft research, `http://research.microsoft.com/en-us/projects/pex/`, 2011.

[24] **Meyer, B.,** Applying 'design by contract', *Computer*, **25(10)** (1992), 40–51.

[25] **Modern Jass,** `http://modernjass.sourceforge.net/`, 2011.

[26] **Nami, M.R. and F. Hassani,** A comparative evaluation of the Z, CSP, RSL, and VDM Languages, *SIGSOFT Software Engineering Notes*, **34(3)** (2009), 1–4.

[27] **Nhat, M.L.,** Contracts for Java: A practical framework for contract programming, `cofoja.googlecode.com/files/cofoja-20110112.pdf`, 2011.

[28] **Object management group - OCL,** `http://www.omg.org/spec/OCL/`, 2011.

[29] **Object management group - UML,** `http://www.uml.org/`, 2011.

[30] **OpenJML,** `http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml`, 2011.

[31] **Python,** `http://python.org/`, 2011.

[32] **Rapide,** The stanford rapide project, `http://complexevents.com/stanford/rapide/`, 2011.

[33] **Reussner, H.R., H.I. Poernomo and W.H. Schmidt,** Reasoning about software architectures with contractually specified components, in: *Component-Based Software Quality* (eds.: A. Cechich, M. Piattini and A. Vallecillo), Lecture Notes in Computer Science, **2693**, Springer, 2003.

[34] **SOFA2,** `http://sofa.ow2.org/`, 2011.

[35] **SPIN** - formal verification, `http://spinroot.com/spin/whatispin.html`, 2011.

[36] **Spivey, J.M.,** *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge University Press, New York, NY, USA, 2008.

[37] **The RAISE Language Group**, *The RAISE Specification Language*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[38] **The wright architecture description language,** `http://www.cs.cmu.edu/~able/wright/`, 2011.

[39] **Unicon,** `http://www.cs.cmu.edu/~UniCon/`, 2011.

[40] **Zhiming Liu, He Jifeng and Xiaoshan Li,** Contract oriented development of component software, in: *Exploring New Frontiers of Theoretical Informatics* (eds.: Jean-Jacques Levy, Ernst W. Mayr and John C. Mitchell), **155**, Kluwer Academic Publishers, Boston, 2004, pp. 349–366.

Gy. Orbán and L. Kozma
Department of Software Technology and Methodology
Faculty of Informatics
Eötvös Loránd University
H-1117 Budapest, Pázmány P. sétány 1/C
Hungary
`o.gyorgy@gmail.com`
`kozma@ludens.elte.hu`