Methods and Tools for Formal Software Engineering

Zhiming Liu[1,⋆] and R. Venkatesh[2]

[1] International Institute for Software Technology
United Nations University, Macao SAR, China
Z.liu@iist.unu.edu
[2] Tata Research and Design Development Centre, Pune, India
r.venky@tcs.com

Abstract. We propose a collaboration project to integrate the research effort and results obtained at UNU-IIST on formal techniques in component and object systems with research at TRDDC in modelling and development of tools that support object-oriented and component-based design. The main theme is an integration of verification techniques with engineering methods of modelling and design, and an integration of verification tools and transformation tools. This will result in a method in which a *correct* program can be developed through transformations that are either proven to be correct or by showing that the transformed model can be proven correct by a verification tool.

1 Formal Software Engineering and the Grand Challenge

The goal of the Verifying Compiler Grand Challenge [7,6] is to build a verifying compiler that

> "uses mathematical and logical reasoning to check the programs that it compiles."

This implies that "a program should be allowed to run only if it is both syntactically and semantically correct" [20]. To achieve this goal, the whole computing community have to deal with a wide range of issues, among which are [2]

1. arriving at automated procedures of abstraction that enables a compiler to work in combination with different program verification tools including testing tools,
2. studying what, where, when and how the correctness properties, i.e. assertions and annotations, are identified and specified,
3. identifying properties that can be verified compositionally, and designing specification notations and models to support more compositional specification, analysis and verification.
4. making tools that are scalable even with specified correctness criteria,

In our view, theories and techniques are a long way from being able to solve the first three problems, and solutions to these problems is obviously vital for dealing with the fourth problem.

[⋆] This work is partially supported by the projects HighQSoftD funded by Macao Science and Technology Fund, NSFC-60573085, NSFC-60673114 and 863 of China 2006AA01Z165.

B. Meyer and J. Woodcock (Eds.): Verified Software, LNCS 4171, pp. 31–41, 2008.

In this position paper, we propose the development *Formal Software Engineering* as a method to develop large software systems using engineering methods and tools that are verifiable. We propose formal modelling of requirements and design, and the automatic generation of code to achieve this. We believe that this effort will contribute towards a solution to the problems stated earlier, in a way that combine techniques and tools of *verification* and those of *correctness by construction* [20].

1.1 The State of the Art in Software Engineering

Software engineering is mainly concerned with the systematic development of large and complex systems. To cope with the required scale traditional software engineers divide the problem along three axes - development phases, aspects and evolutions. The development phases are - Requirements, Design and Implementation. Each development phase is divided into different aspects, such as:

- static data model, control flow and operations in the requirements phase;
- design strategies for concurrency, efficiency and security in the design phase. These strategies are commonly expressed as design patterns [3]; and
- databases, user interface and libraries for security in the implementation phase.

The third axis is that of system evolution and maintenance [9,8] where each evolutionary step enhances the system by iterating through the requirements - implementation cycle. Unfortunately all aspects are specified using informal techniques and therefore this approach does not give the desired assurances and productivity. The main problems are:

- Since the requirements description is informal there is no way to check for its completeness, often resulting in gaps.
- The gaps in requirements are often filled by ad-hoc decisions taken by programmers who are not qualified for the same. This results in rework during testing and commissioning.
- There is no traceability between requirements and the implementation, making it very expensive to accommodate changes and maintain the system.
- Most of the available tools are for project management and system testing. They are not enough to ensure the semantic alignment of the implementation w.r.t a requirements specification and semantic consistency of any changes made in the system.

1.2 The State of the Art of Formal Methods

Formal methods, on the other hand, attempt to complement informal engineering methods by techniques for formal modelling, specification, verification and refinement. They have been extensively researched and studied. A range of semantic theories, specification languages, design techniques, and verification methods and tools have been developed and applied to the construction of programs of moderate size that are used in critical applications. However, it is still a challenge is to scale up formal methods and integrate them into engineering development processes for the correct construction and maintenance of software systems, due to the following problems:

- Each development is usually a new development with very little reuse of past development.
- Because of the theoretical goal of completeness and independence, refinement calculi provide rules only for a small change in each step. Refinement calculi therefore do not scale up in practice. Data refinement requires definition of a semantic relation between the programs (their state space) and is hard to be applied systematically.
- Given low level designs or implementations it is not easy for software engineers to build correct and proper models that can be verified by model checking tools.
- There is no explicit support for productivity enhancing techniques such as component-based development or aspect-oriented development.

We also observe that verification techniques and tools (e.g. model checking, SAT solving, etc.) have only been relatively effective only in the development of hardware systems. An integration of such methods with software development is highly required by the manufacturers of critical and embedded software (avionics, telecom, public transport, etc.). However, the sophisticated nature of software (complex data structures, recursion, multithreading) poses challenging theoretic and practical problems to the developers of automatic analysis and verification methods.

Both formal methods and the methods adopted by software engineers are far from meeting the quality and productivity needs of the industry, which continues to be plagued by high development and maintenance costs. Complete assurance of correctness requires too much to specify and verify and thus a full automation of the verification is infeasible. However, recently there have been encouraging developments in both approaches. The software engineering community has started using precise models for early requirement analysis and design [18]. Theories and methods for object-oriented, component-based and aspect-oriented modelling and development are gaining the attention of the formal methods community. There are attempts to investigate formal aspects of object-oriented refinement, design patterns, refactoring and coordination [12].

1.3 Aims and Objective

The aim of this project is to combine the strengths of software engineering techniques and formal methods thus enabling the development of systems that have the assurances possible due to formal methods and productivity and scale-up achievable by methods adopted by software engineers.

We will focus on the development of a theory of modelling (or specification), analysis and refinement of component and object systems, and a toolset that integrates two kinds of complementary tools: tools for analysis (model checkers and theorem provers) and tools for correctness preserving transformations, including design patterns and domain specific transformations. We will study and verify the correctness of the transformations, aiming at verified designs transformations to scale up formal methods by

- exploiting standard design patterns and strategies existing in large applications, even across applications;
- providing verified design patterns and strategies to reduce the burden on (automated) proofs; and
- proving functionality correctness only at the specification stage.

We can also think this is about the development of a CASE tool that is supported by a formal theory and combines model transformation and model verification.

2 Formal Modelling of Complex Systems

This section gives a brief outline of the technique and solution to be investigated by this project. The techniques are explained using a simple example of a library system, that maintains a collection of books. Members belonging to the library borrow and return books. In order to keep the explanation simple and readable we have not been rigorous in the specification of the library system. For more formality, we refer the reader to the paper [14]. Also in [17], a Point of Sale (POST) system was formally developed, including a C# implementation.

2.1 Requirements Modelling

For an object system or a component, the development process begins with the specification of functional requirements. Functional requirements of a system consists of three aspects: the state, a set of operations through which external agents may interact with the system, and a set of global properties that must be satisfied by the state and operations. This can be represented as a triple $RM = \langle S, O, I \rangle$ where S is a model of the state, O is a set of operations that modify the state and I is a set of global invariants. Each operation in O is expressed as a pre- post-condition pair [12]. A requirements model is consistent if each operation in O is consistent with the state model and preserves the global invariant. The model can be further enhanced by adding descriptions of interaction protocols with the environment [5], timing aspects, features of security, etc. A multi-view and multi-notation modelling language, such as a formalized subset of the Unified Modelling Language(UML) [19], can be used to specify this model and analyzed for inconsistencies using model-checking techniques as demonstrated in [22]. The analysis can be carried out incrementally, a small number of use cases at a time that only involve a small number of domain classes [14]. This is obviously important to development of tool support to the analysis.

Library requirements. The state space of the library system is represented by the tuple $\langle Shelf, Book, Member, Loan : Book \times Member, isIn : Book \times Shelf \rangle$ where, *Book*, *Member* and *Shelf* are set of books, members and shelves in the library. *Loan* is a set of tuples representing the books that have been currently loaned to members. The association *isIn* is a set of tuples representing books that are currently on some shelf. This state space corresponds to a UML diagram and can be formalized as a class declartion section of an OO program [14,13].

The set of operations will be $\{\mathit{Borrow}(\mathit{Member}, \mathit{Book}), \mathit{Return}(\mathit{Member}, \mathit{Book})\}$. These operations are identified from the use cases [14]. The *Borrow* operation can be described as

$$\begin{array}{ll}
\mathbf{signature}: & \mathit{Borrow}(S, S' : \mathit{State}, b : \mathit{Book}, m : \mathit{Member}) \\
\mathbf{pre-condition}: & \neg\exists m_1 : \mathit{Member} \bullet \langle b, m_1\rangle \in S.\mathit{Loan} \\
\mathbf{post-condition}: & S'.\mathit{Loan} = S.\mathit{Loan} \cup \langle b, m\rangle \wedge S'.\mathit{isIn} = S.\mathit{isIn} - \langle b, s\rangle
\end{array}$$

Return can be defined similarly.

A sample invariant is *BookInvariant*, which states that every book in the library is either on the shelf or loaned to a member. This can be stated as follows.

$$\begin{array}{lll}
\mathit{BookInvariant}(S : \mathit{State}) \overset{def}{=} & & \forall b : S.\mathit{Book} \bullet \exists m : \mathit{Member} \bullet \langle b, m\rangle \in S.\mathit{Loan} \wedge \\
& & \neg\exists s : \mathit{Shelf} \bullet \langle b, s\rangle \in S.\mathit{isIn} \\
& \vee & \neg\exists m : \mathit{Member} \bullet \langle b, m\rangle \in S.\mathit{Loan} \wedge \\
& & \exists s : \mathit{Shelf} \bullet \langle b, s\rangle \in S.\mathit{isIn}
\end{array}$$

Details on the formalisation of a use-case model and its consistency relation with a class model (i.e. the state space) can be found in [14].

2.2 Design

Design involves transforming the requirements model of a system to a model with design details, by design strategies or patterns as functional decomposition and object or class decomposition. This model is still platform independent models (PIM) [13].

In a later stage, the PIM is transformed to a model of a platform or a family of platforms (PDM) with desired non-functional properties such as - support for concurrent or parallel execution, performance and usability. The platform may be modelled by a tuple, $\langle S_p, O_p\rangle$ where S_p is a meta-model of the platform state and O_p is a set of platform operations which maybe combined using a set of available operators.

Given a PDM, a system is designed by transforming the PIM state, *S* to a design state S_d that is an instance of the PDM state S_p and transforming each operation $o \in O$ to an operation o_d, which is expressed as a composition of operations in O_p. The design step also specifies a set of design invariants I_d that the design operations must preserve. Thus the design model is a triple, $\langle S_d, O_d, I_d\rangle$ where O_d is the set of all transformed operations and the design process consists of two transform functions $\langle T_s, T_o\rangle$ where $T_s : S \rightarrow S_d$ is the state transformation function and $T_o : O \rightarrow O_d$ is the operations transformation function. A design is correct if the two transformation functions are consistent that is the diagram in Figure 1 commutes and the design operations preserve the design invariants.

Library design. To simplify the presentation assume the library requirements model to consist of $\langle S_r, O_r\rangle$, where S_r is a set of class and association names and O_r the operations:

$$S_r = \{Shelf, Book, Member, Loan, isIn\},\ \ O_r = \{Borrow, Return\}$$

$$\begin{tikzcd}
s1 \arrow[r, "o"] \arrow[d, "Ts"'] & s2 \arrow[d, "Ts"] \\
sd1 \arrow[r, "Td(o)"'] & sd2
\end{tikzcd}$$

Fig. 1. Design Transformations

Further associate each operation *op* with the set of state objects *obj*(*op*) it accesses, because for concurrency only the object being accessed is relevant and not the details of the modifications to the object. So

$$\mathit{obj}(\mathit{Borrow}) = \mathit{obj}(\mathit{Return}) = \{\mathit{Loan}, \mathit{isIn}\}$$

Assume the library system is to be implemented on a platform where multiple processes run the library operations and all of these refer to a set of shared objects. The design model will be $\langle S_d, O_d \rangle$, where

$$S_d = S_r \cup \{s_l, s_i\}, \qquad O_d = \{Borrow_d, Return_d, P, V\}$$

where s_l and s_i are semaphores corresponding to `Loan` and `isIn` objects, and P and V are the semaphore operations. The operations in the design are defined by a sequence of semaphore operations and objects accessed. Thus,

$$\mathit{obj}_d(\mathit{Borrow}_d) = \mathit{obj}_d(\mathit{Return}_d) = [s_l; s_i; \mathit{Loan}; isIn; s_l; s_i]$$

This design guarantees - correctness, mutual exclusion and deadlock freedom.

Instead of designing each library operation individually we can write two design transformation functions for the library design as follows

$$\begin{array}{lll}
T_{ls} & \overset{def}{=} S_r \cup \{S_i \mid S_i \text{ is a semaphore for } s_i \in S_r\} & \\
T_{lo}(\mathit{op}) & \overset{def}{=} [P_1; \cdots; P_k; a_1; \cdots; a_k; V_1; \cdots; V_k] & \text{if } \mathit{op} \text{ is realized by the sequence} \\
 & & a_1; \ldots; a_k \text{ of accesses to } s_1, \ldots, s_k
\end{array}$$

We can prove the correctness of the transformation as required by figure 1. Also, mutual exclusion and deadlock freedom can be guaranteed. Since the design has been implemented as a transformation we do not have to prove correctness of the design specification for each operation, instead we prove correctness of the transformation.

Design Patterns. Different systems adopt similar design transformation functions. Therefore the process of formal design can be scaled up by abstracting away from individual design transformation functions to a design pattern. A design pattern is a meta-function that maps a requirements model to a design transformation function for that requirements model. A design pattern is correct if the mapped design functions are correct as described above. Design patterns can be proved correct independent of the requirements model making them scalable. In the presence of design pattern a design step will involve selecting and applying the appropriate design patterns.

For the Library example, the design strategy of imposing a total order on the semaphores can be abstracted out into a transformation function. The transformation function takes the total order and a requirements specification as input and transforms the requirements of an arbitrary system into a corresponding design specification. Thus a design pattern for databased applications that supports multiple users and guarantees mutual exclusion and deadlock freedom consists of two transformation functions in the form of T_{ls} and T_{lo} of the library system.

MasterCraft [1] implements a few such design patterns for some select platforms and design strategies. MasterCraft however does not support formal specification and verification of these design patterns. If implemented as a design pattern the atomicity preservation and deadlock freedom will not have to be proved for each application of the transformation. All we need to show is that for a given application there the given total order on objects includes all the objects that are referred to by any of the operations of the system. We believe that is achievable in the framework of *r*COS.

*r*COS also provide a general refinement calculus for correctness preserving transformation between PIMs. General software design patterns, such General Responsibility Assignment Software Pattern (GRASP) [10], are formalized as refinement rules in *r*COS [13]. Here we use UML to represent some of the refinement rules in *r*COS:

Functional Decomposition: This is also known as the *expert pattern* which allows us to delegate that part of the functionality of method N in Figure 2, which only refers to attributes x of class M, to the expert M of information x.

Class Decomposition: Figure 3 shows how we can decompose a complex class into a number of related but simpler classes. Figure 4 represents another way of class decomposition. Class decomposition rules are known by OO engineers as High Cohesion Pattern.

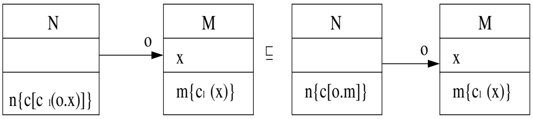

Fig. 2. OO Functional Decomposition

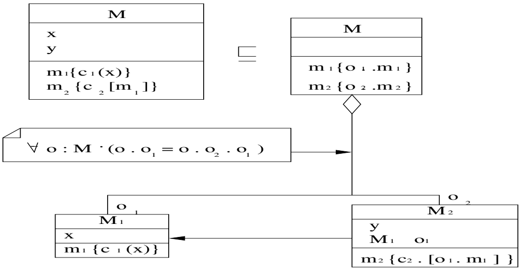

Fig. 3. Class Decomposition 1

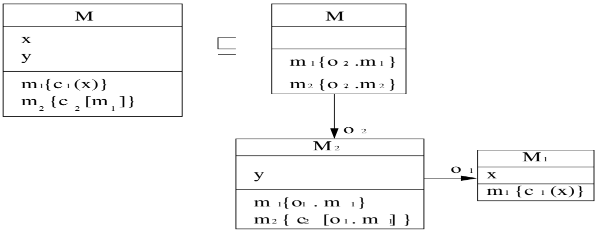

Fig. 4. Class Decomposition 2

Low Coupling: The *Low Coupling Pattern* represented in Figure 5 allows us to obtain the design in Figure 4 from the design in Figure 3.

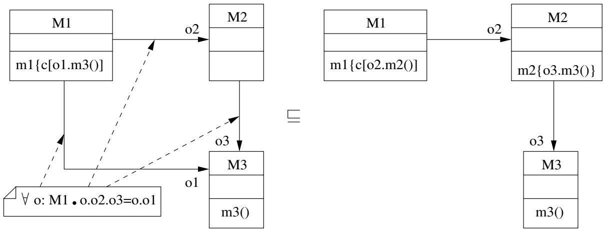

Fig. 5. Low Coupling

More design patterns and pattern-directed refactoring are also studied and applied to the case study POST [17]. We will extend MasterCraft by adding the implementations of these rules.

3 Research Problems

The previous section presented an overview of a proposed method for formal development of large scale systems. To realize this method, we first need to define it more formally. We aim at a logically sound and systematic method (that we are tempted to call a *formal engineering method*) and tools that themselves are provably correct for supporting the method. The method includes:

1. *A language and a logic for specifying and reasoning about a system at different levels of abstractions.* The main task is to develop a notation for describing each aspect of correctness of a model. This will allow a developer to split a model of a system into several aspects making it more manageable. This is important for tool development too. The notation for a particular aspect should be expressive enough for describing all the concerns about that aspect. However, overlapping features among different notations should be kept to a minimum else, problems of inconsistency and integration will become overwhelming[1].

[1] This is a serious problem in the application of UML.

The logic should provide a sound link among the different notations to deal with the problems of model consistency and integration. It should support compositional reasoning about the whole model by reasoning about the sub-models of the aspects. Different verification techniques and tools maybe applied to models of different aspects of functionality, interaction and structure of the system.

2. *A Language and logic for specifying the transform functions and reasoning about correctness.* The language should preferably be composable. That is, it should be possible to specify various design transformations independently and compose them to get a design from requirements. The techniques and tools will include formally proved pattern-directed transformations of specifications to scale up the classical calculi of refinement. We will also investigate the use of model checking and static analysis techniques and tools for consistency and analysis of properties of models. For specification and analysis of coordination among components, simulation techniques and tools can be used. Transformation of different sub-models may need different verification techniques and tools. Data refinements will be realized by structural transformations following design patterns that are scaled up from object-oriented design.
3. *Automatic code generators that implement the implementation functions for various platforms.* Refactoring transformation of designs and implementations will be studied and implemented in the tool support.
4. *Techniques and tools for domain-specific languages and their programming* (such as web-based service and transaction system based on internet).

The main theme of the project is to integrate formal verification techniques and tools with design techniques and tools of model (or specification) transformations. Verification and transformation will work complementary to ensure the correctness of the resultant specification. The design techniques and transformation tools are essential in the development to transform the requirements specification to a model that is easy to be handled with the verification techniques and tools. The design and transformation have to be carried interactively between the designer and the tool. Verification tools can be also invoked during a transformation [21].

This project will be conducted in a close collaboration between UNU-IIST and TRDDC. UNU-IIST is particularly strong in theories and techniques for program modelling, design and verification, and TRDDC is the largest industry research development and design centre in India. We will investigate how the research results at UNU-IIST in theories and techniques of program modelling, design and verification can be used in the design of software development tools at TRDDC. A separate position paper by UNU-IIST is also presented at this conference [2].

4 Related Work at UNU-IIST and TRDDC

TRDDC and UNU-IIST have been approaching the above problem from two different ends. TRDDC has expertise in software engineering techniques and has been researching this area for several years now. These efforts have resulted in MasterCraft [1], a tool that generates code for different platforms from design specifications. Current research activities at TRDDC include graph-based languages for specifying requirements [22] and transformations. The requirements group has successfully used model checking to verify correctness of requirements of a few projects. The work on transformation specifications has resulted in a proposal as a standard in response to an OMG request. The proposal is in an advanced stage of acceptance.

UNU-IIST has been working on formalizing object-oriented development. This work has resulted in a relational model for object-oriented design and an associated refinement calculus [13]. The refinement calculus supports incremental and iterative development [14]. The model is current being extended to support component-based development [5]. Initial progress have been made in experimental development of tool support [11,16]. Promising results have been achieved in unifying different verification methods [4,15].

5 Summary

We believe that we need to advance theories, tools and experiments for both verification and design, and to scale them up to meet business and engineering projects need. For this, we propose component-based modelling and design by transformations so that a software designer can

- apply verified model transformations or define a transformation and verify it after applying it,
- model method bodies (hopefully, the methods now are simple)
- generate proof-carrying code from target model

References

1. Mastercraft. Tata Consultancy Services, `http://www.tata-mastercraft.com`
2. Aichernig, B.K., He, J., Liu, Z., Reed, M.: Theories and techniques of program modelling, design and verification. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments (VSTTE 2005). LNCS, vol. 4171, pp. 291–300. Springer, Heidelberg (2008) (this volume)
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addlison Wesley, London (1994)
4. He, J.: Link simulation with refinement. In: Proc. of The 25th anniversary of CSP (2004)
5. He, J., Li, X., Liu, Z.: Component-based software engineering – the need to link methods and their theories. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 72–97. Springer, Heidelberg (2005)
6. Hoare, A.C.R., Misra, J.: Verified software: Theories, tools and experiments. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 1–18. Springer, Heidelberg (2008) (this volume), `http://vstte.ethz.ch/`
7. Hoare, C.A.R.: The verifying compiler: A grand challenge for computer research. Journal of the ACM 50(1), 63–69 (2003)
8. Joseph, M.: Formal techniques in large scale software engineering. In: Keynote at IFIP Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE), Zurich (October 10-13, 2005), `http://vstte.ethz.ch/`

9. Kruchten, P.: The Rational Unified Process – An Introduction, 2nd edn. Addison-Wesly, London, UK (2000)
10. Larman, C.: Applying UML and Patterns. Prentice-Hall, Englewood Cliffs (2001)
11. Li, X., Liu, Z., He, J., Long, Q.: Generating prototypes from a UML model of requirements. In: International Conference on Distributed Computing and Internet Technology (ICDIT2004), Bhubaneswar, India. LNCS, Springer, Heidelberg (2004)
12. Liu, Z., He, J., Li, X.: Contract-oriented development of component systems. In: Proceedings of IFIP WCC-TCS2004, Toulouse, France, pp. 349–366. Kluwer Academic Publishers, Boston (2004)
13. Liu, Z., He, J., Li, X.: rCOS: A refinement calculus for object systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 183–221. Springer, Heidelberg (2005)
14. Liu, Z., He, J., Li, X., Chen, Y.: A relational model for object-oriented requirement analysis in UML. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 641–664. Springer, Heidelberg (2003)
15. Liu, Z., Ravn, A.P., Li, X.: Unifying proof methodologies of Duration Calculus and Linear Temporal Logic. Formal Aspects of Computing 16(2) (2004)
16. Long, Q., Liu, Z., He, J., Li, X.: Consistent code generation from uml models. In: Australia Conference on Software Engineering (ASWEC), IEEE Computer Scienty Press, Los Alamitos (2005)
17. Long, Q., Qiu, Z., Liu, Z., Shao, L., He, J.: POST: A case study for an incremental development in rCOS. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 485–500. Springer, Heidelberg (2005)
18. Mellor, S.J., Valcer, M.J.: Executable UML: a foundation for model-driven architecture. Addison-Wesley, Reading (2002)
19. OMG. The Unified Modeling Language (UML) Specification - Version 1.4, Joint submission to the Object Management Group (OMG) (September 2001) http://www.omg.org/technology/uml/index.htm
20. Pnueli, A.: Looking ahead. In: Workshop on The Verification Grand Challenge 2005 SRI International, Menlo Park, CA(February 21–23)
21. Rushby, J.: Integrating verification components. In: Keynote at IFIP Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE), Zurich, (October 10-13, 2005), http://vstte.ethz.ch/
22. Shrotri, U., Bhaduri, P., Venkatesh, R.: Model checking visual specification of requirements. In: International Conference on Software Engineering and Formal Methods (SEFM 2003), Australia, 3003., pp. 202–209. IEEE Computer Society Press, Los Alamitos