

Harnessing Theories for Tool Support*

Zhiming Liu¹, Vladimir Mencl^{1,2}, Anders P. Ravn³, and Lu Yang¹

¹*United Nations University
International Institute for Software Technology
{lzm,yanglu,mencl}@iist.unu.edu
http://www.iist.unu.edu/*

²*Department of Software Engineering, Charles University, Czech Republic
mencl@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz*

³*Department Computer Science, Aalborg University, Denmark
apr@cs.aau.dk*

Abstract

Software development tools need to support more and more phases of the entire development process, because applications must be developed more correctly and efficiently. The tools therefore need to integrate sophisticated checkers, generators and transformations. A feasible approach to ensure high quality of such add-ins is to base them on sound formal foundations. In order to know where such add-ins will fit, we investigate the use of an existing successful commercial tool and identify suitable places for adding formally supported checking, transformation and generation modules. The paper concludes with a discussion of feasibility of developing the proposed add-ins and of the conditions for such extensions to be actually used.

Keywords: *Software development tool, software process, formal methods, tool design.*

1. Introduction

Software development is a huge industry, and there is an accelerating demand for more and more software to implement all kinds of applications which make the world convenient for us. That is, when the software works as intended. And it is not becoming easier to develop quality software, because the applications have to implement more and more complex functionalities, and often it has to be done on very dynamic, distributed and heterogeneous platforms. Software developers can no longer produce the required applications just by assem-

bling code, even when they are supported by extensive libraries with many utilities. There is a need for tools that support the development process, in such a way, that the developers can focus on modelling the application, and the tools synthesize the application for a specific platform and provide efficient test and validation environments.

Such tools that support a model based development discipline are becoming available. They typically rely on design paradigms from the object oriented world, because with the Unified Modeling Language (UML) [11], there is a consensus on notations for the different aspects of a software system: use cases for requirements, class diagrams for structural design, sequence and state diagrams for protocols and behaviors, just to mention some of the most used constituents of the language. However, a tool is not only a set of structured editors and a repository for pieces of design documents. In order to be really useful, it must support the synthesis process by providing code generators which produce the implementation, either automatically, or semi-automatically, with the assistance of programmers that build the detailed code. A tool must also support linking to libraries, linking to independently developed subsystems, like databases, and linking to the platforms on which the application is going to run. Furthermore, an advanced tool will allow several views of the models for the different roles of developers, for instance the architects need to have different views than the component developer, or the version manager. In the development process, these different actors need support for checking consistency of the operations they perform, for generating tests, for transforming one element to another, etc. Tools are in themselves complex software systems, and unless they are developed with care, one may fear that they will suffer from "feature interactions" — this

*This is a short version of [21]. This work is partially supported by the project HighQSoftD funded by Macao Science and Technology Development Fund.

nice term was coined in the telecommunications industry [2] for inconsistent behaviors caused by conflicting functionalities.

Since tools are very important, there is a considerable amount of research that addresses the issue of building better tools by using a formal foundation for the involved languages and processes [16, 19]. There is also research that aims at improving the facilities of existing tools by integrating formally founded sub-tools, in particular model checkers, into existing tools [6]. However, we have decided to use some time to take a fresh look at an existing successful tool and investigate in detail, where formal methods-based reasoning would potentially improve the capabilities of the tool. In the process, we put emphasis on keeping the tool consistent with its intended use. The specific contributions of this paper are:

- A detailed description of the processes and procedures supported by a specific tool.
- An analysis of suitable places for consistency checkers, verification of properties, transformations and generators based on formal interpretation of the input and output models of elements stored in the tool.

The tool that we examine is MasterCraft [29]. We have selected it, because it has extensive coverage of the whole software development life-cycle, from requirements gathering and analysis, through early design stages to implementation and testing, with support for deployment and maintenance. Additionally, it is known to be successfully used in its intended application area: Web Systems. Finally, it plays a major role that the producer of MasterCraft, Tata Research Development and Design Centre (TRDDC), generously have permitted us to inspect the tool in detail.

With respect to formal methods, we have recently developed a rather rich and mature formalism that models static and dynamic features for component based systems. This theory, which is called rCOS [13, 4] and is based on the UTP framework [15], forms the basis for our analysis of what should be feasible in a further enrichment of a tool like MasterCraft.

The theory of rCOS shares the idea with Circus [3] in their attempts to combine event-based models and state-based models. However, rCOS has an extensive theory about the consistency among different views of models and provides precise characterizations of the notions and composition operations of provided and required interfaces, contracts, protocols, component implementations, component publications, and component coordinations. rCOS also has a direct Java-like specification language and deals with inheritance, reference

types and dynamic binding. The stream calculus [14] and Reo [12] are also quite mature formal theories for component-based development. They are both channel based models in that the output traces of a component are defined to be functions of the infinite input traces.

Overview. The following Section 2 introduces MasterCraft and gives a detailed description of its use. Section 3 identifies the different processes in the use of MasterCraft where add-ins based on formal reasoning may improve the process. For each there is a concrete suggestion for the functionality of the add-in. Finally Section 4 concludes and discusses the findings and the conditions for success, if the outlined tool enhancement is implemented. There are also pointers to further work.

2. Industrial tool support: MasterCraft

MasterCraft [29] is developed by TRDDC to support efficient development of software system. In MasterCraft, different activities at different stages of development are performed by project participants in different roles. We see this distinction as very important, as it allows us to define at which point in the development process should various models be *produced*, and different kinds of *manipulation*, *analysis*, *checking* and *verification* be performed, with different tools. We make the particular roles responsible for assuring the correctness of the resulting software system.

In this section, we describe the software development lifecycle in MasterCraft by elaborating on the roles and their responsibilities. In Section 3, we will discuss formal method tools to support the roles in their activities.

2.1. Roles in software development

In MasterCraft, the members of the development team are assigned different *roles* in the development process: *Administrator*, *Analysis Modeler*, *Design Modeler*, *Construction Manager*, *Construction Programmer*, *Model Manager*, and *Version Manager*. Each role gives different rights to access the project artifacts.

In this paper we will focus on roles who handle the modelling tasks and construction tasks, such as contributing new functionality (horizontal refinement) or lifting the model to a more concrete level (vertical refinement), instead of the more technical roles like *Administrator*, *Version Manager* and *Model Manager* who are more concerned with managing aspects of the project.

Modelling tasks. Firstly, the *Analysis Modeler* starts work on a component and develops the analysis model based on the textual requirements. The analysis model consists of conceptual class diagrams, use case diagrams and behavioural models like sequence diagrams and state machine diagrams. The *Analysis Modeler* may iterate over this model, creating a new refined model based on the original analysis model.

In the next step, the *Design Modeler* refines the analysis model into a design model which will go into the details and allow the generation of executable code. The design model consists of object diagrams, object sequence diagrams (interactions by method invocation of objects participating in a use case) and object state machine diagrams (defining behaviour of objects). This lays the groundwork for the construction tasks.

Construction tasks. The *Construction Manager* is the key role responsible for construction tasks. The *Construction Manager* starts by exporting the design model and invokes code generation tools, which generate code templates for all the classes. Subsequently, the *Construction Manager* assigns coding of these operations to *Construction Programmers* and the *Construction Programmer* will code and unit test the assigned operations. After receiving code for all the tasks assigned to different *Construction Programmers*, the *Construction Manager* integrates the code together.

Example. We will illustrate software development in MasterCraft on the case study of a simple Point-Of-Sales (POS) System originally proposed by Larman [18]. The system supports management of sales, inventory, and payments. The inventory maintains a catalog of *product specifications*, and the amount of each product available. The system also keeps records of sales, consisting of a number of items, determined by the product specification and the amount sold. The system also maintains the payments, associated with the sales.

Fig. 1 shows the classes created in the POS case study and their partitioning into components. We also illustrate the behavioral analysis models created for the *SalesManagement* component with the sequence diagram shown in Fig. 3 (a) and state machine shown in Fig. 3 (b). The design model created based on the analysis model adds new elements needed for the design, such as operations of classes. We illustrate this on the *design class diagram* shown in Fig. 2. In addition, design-level behavioral diagrams now model the interaction among the classes and we illustrate this on the *design sequence diagram* shown in Fig. 3 (c).

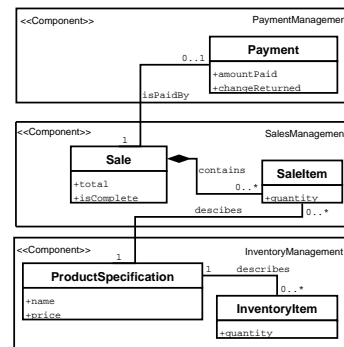


Figure 1. POS: analysis class diagram

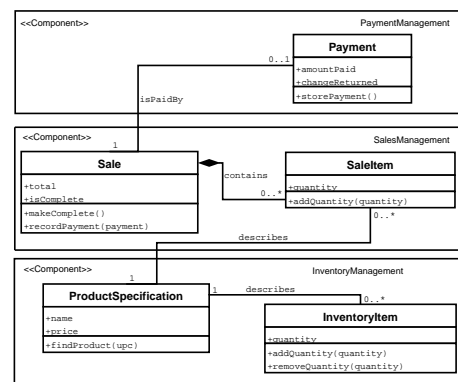


Figure 2. POS: design class diagram

3. Desirable Formal Support

In the software development process currently established in MasterCraft, certain verifications of models are already employed; however, there is potential for significantly advancing the verification support. The current checks focus on checking structural consistency of models, and managing such mandatory checks throughout the software development process. For example, MasterCraft requires to use the function *Validate User Model* to check structural consistency of a design model, before either using the model for code generation, or releasing the model into the shared pool.

The model checking tool SAL is also integrated into MasterCraft. In principle, some properties of models can be verified by this tool. Indeed, an invariant property that is required to be preserved by the use cases of an analysis model can be checked. For the POS System, the properties that *when a sale is completed and paid, the total of a sale must be equal to the amount paid minus change returned* is an invariant of the analysis model. This checking is possible, because there is a

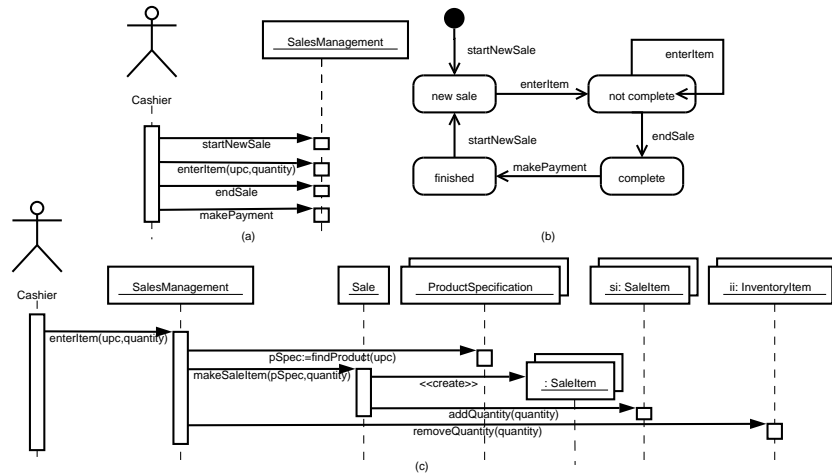


Figure 3. POS: (a) analysis sequence diagram, (b) analysis state diagram and (c) design sequence diagram

formal semantics for the state diagram of the use cases. However, the use of SAL in MasterCraft has so far been limited to analysis models. The reasons are

1. there is a lack of formal semantics for the design model,
2. there is very limited use of the behavioral and interaction parts of the analysis model in the design and implementation models,
3. SAL (or any other model checking tools) cannot in practice verify detailed design and implementation models, without the support of abstraction, compositional verification and stepwise transformations in the design process.

There is thus ample room for improvement based on research on semantics of object and component systems [13, 4], together with their applications in formalizing constructs of UML [20, 5, 30, 23, 24, 31]. It should be feasible to support more design activities and more comprehensive verifications. In the rest of this section, we outline how these may be embodied into MasterCraft to serve the needs of the different development roles.

3.1. Component modelling

Before components are assigned to an Analysis Modeler, the application components must be identified and the primary architectural decisions must be made. Even agreeing on the list of components and their interface dependencies is a serious architectural decision. Such a highly abstract architecture model can be

supported with component diagrams of UML 2.0 [11]. When deciding on partitioning the application into components in large and complex projects, a model may help to properly define the components at the syntactic level (Provided and Required Interfaces of Components) and at the semantic level (Contracts of Interfaces).

In the framework of rCOS [4], an *interface* of a component declares a list of state variables with their types and list of operations (or methods) with their parameters and types. A *contract* of the interface defines the functionality by specifying a precondition and a postcondition for each operation, and by specifying a protocol as a set of finite traces (sequences) of operations. Compositions are defined in rCOS for plugging the provided interface of one component into the required interface of another, parallelly composing two disjoint components, renaming and hiding operations of interfaces, and feedback of a component.

We introduce processes into rCOS in [4]. Unlike a component that is passively waiting for a client to call its provided services, a process is active and has its own control on when to call out or to wait for a call to its provided services. For simplicity but without losing expressiveness, we assume a process like a Java thread does not provide services and only calls operations provided by components. Therefore, processes can only communicate via shared components. The composition of two processes is defined similarly to the alphabetized parallel in CSP [27].

The separation of the concerns about the functionality and interaction aspects becomes more important when we compare contracts and when we assemble or

substitute components according to their provided and required interfaces. For this, we can check the interaction compatibility and functional compatibility separately, possibly with different tools, such as SAL or FDR [27]. If the protocols are only regular languages (i.e., specified by regular expressions), the checking can be completely mechanized. However, functional compatibility requires that the provided functionality refines the required functionality. Functionality refinement is defined by logic implication and thus may need theorem provers or model checkers [26, 25].

The extension to MasterCraft discussed above indicates the possible need of a role, say *Architecture Modeler* which would take a part of the responsibilities of *Administrator*. This new role is responsible for creating and analyzing the architectural or component model before assigning components to Analysis Modelers. The architectural models would be then helpful for the Version Manager and Construction Manager in dealing with problems like consistency between versions of a component, substitutability of components, and assembly of components.

3.2. Analysis modelling

The aim of the Analysis Modeler is to produce an analysis model for the component assigned to him, based on the requirement documents kept outside MasterCraft. In general, an analysis model consists of (*conceptual*) class diagrams, use case diagrams, use case descriptions or specifications, sequence diagrams and state diagrams of use cases [20].

Simple use cases, such as *MakePayment* in the POS example, can be defined as a one step operation by a precondition and postcondition. An invariant can be specified in OCL and checked by MasterCraft. In general, a use case, such as the *BuyItems* use case of POS, has a number of atomic steps that we call *use case operations*, and use cases depend on and interact with each other. (For instance, the *BuyItems* use case invokes the *MakePayment* use case). More extensive use of activity diagrams, sequence diagrams and state diagrams will document such dependencies. Then a challenging issue is how to describe and check the *consistency* among the diagrams of the whole analysis model. A precise semantics of each diagram and the integration is a major step towards dealing with this problem.

In [20], we give formal definition of a requirements model and the consistency relation between the class and use case models, and also define a refinement relation between analysis models, which supports incremental requirements capture and analysis. By being incremental, we roughly mean that the analysis model is

produced and analyzed by adding use cases and their required classes and associations step by step.

Semantics of sequence diagrams is studied in [5]. Based on these, an initial version of a tool, called AutoPA1.0, is developed for automatic generation of executable prototypes of analysis models [30]. This initial version shows the promising of checking consistency and validating use cases. Also, algorithms are developed for code generation from class diagrams and sequence diagrams [23]. They may be very useful for rapid prototyping such that the informal user expectations can be validated, and they may even in less computationally demanding cases be used in the construction stage.

Notice that in AutoPA1.0, the generated code includes either specification or implementation of method bodies, depending on the amount of details modelled in the sequence diagram. Furthermore, upon integration into MasterCraft, the semantic definitions will allow to employ other existing verification tools for more extensive analysis of the analysis model if required for the application. This includes checking general safety properties, and additional properties optionally specified by the Analysis Modeler.

3.3. Model transformations in design

The Design Modeler takes the analysis model of a component produced and validated by the Analysis Modeler, and produces a design model (we call it the *Final Design Model*) that is ready for code generation. This model is a refinement of the analysis model, and consists of an *object diagram* (or packages of object diagrams), *object sequence diagrams* and *object state diagrams*. The object diagram gives detailed declarations of attributes and method signatures of each class, and specifies the inheritance, association and dependency relations among classes (cf. Fig.2). An object sequence diagram defines a use case operation in terms of interactions (method invocations) among associated objects that participate in the use case (cf. Fig. 3 (c)). An object state diagram defines the behavior of an (important) object. In MasterCraft, the object diagram is used for automatic generation of the *code template*. The sequence diagrams and state diagrams are however only used for guiding the Construction Programmers to program the bodies of the methods.

For a realistic application, the final design model obviously cannot be produced in one step from the component analysis model. Quite contrary, a stepwise, incremental, and iterative design process, such as the Rational Unified Process (RUP) [17], is usually adopted. In such a process, each subsequent version should be pro-

duced from its preceding version by applying a transformation that preserves already specified properties. In fact, the sequence diagram in Fig. 3 (c) is created by a number of applications of the design patterns for object responsibility assignment. The design class diagram in Fig. 2 is constructed from the analysis class diagram in Fig. 1 and the sequence diagrams for all use cases.

The theoretical frameworks supporting such a design process are usually referred to as *correctness by design*. rCOS supports *correctness by design*. Other example frameworks of *correctness by design* include JML [19] and Alloy [16]. The widely known Model Driven Architecture (MDA) is a possible framework for correctness by design.

In rCOS, correctness preserving transformations are characterized by *refinement rules*. These are generally logical implications, which are in general undecidable. Therefore, theorem provers or proof checkers are the expected tools to support this approach. The good news is that proved design patterns [9] and refactorings [8] can be programmed in a transformation language such as QVT [10]. For the patterns *expert*, *class decomposition*¹, and *attribute encapsulation*, a proof of their correctness with respect to refinement has already been given in [13] and applied to the development of the POS system [24]. Refactoring rules are proved in rCOS in [22] and applied in the case study POS in [24]. Extending MasterCraft with implementations of these design patterns and refactoring rules will enhance the support provided to the Design Modeler in producing good designs. The future work of correctness preserving transformation is listed in [31].

However, for some of the transformations, the refinement holds only when additional conditions are satisfied by the resulting model. These properties can sometimes be proved by model checking the resulting model. Some of them, however, have to be carried on to the resulting code, and checked with static analysis tools or by run time monitoring. Some properties, such as fairness properties and timing properties, can only be verified when the target platform is known. In these cases, the transformation would annotate these conditions in the transformed model elements.

3.4. Construction & code generation

The current version of MasterCraft generates code templates, and the sequence diagrams and state diagrams in the final design model are used as an informal guide to the Construction Programmer to program the bodies of the class methods.

¹More widely known as High Cohesion and Low Coupling Patterns.

With semantics of sequence diagrams in [5] and the algorithms in [23], we can extend the MasterCraft code generator to generate method invocations in the body of a class method with correct flow of control (i.e., the conditional choice and loop statements).

Furthermore, the Design Modeler can specify *class invariants* and the functionalities of each class method in terms of its precondition and postcondition about the change of the object state for methods and classes added in the detailed design. This can be written either in OCL or in Spec# assertion commands [1] and attached to the class. Alternatively, the precondition and postcondition of a method can be shown in the sequence diagram (but this might make the sequence diagrams unreadable). Then it is possible for these conditions to be automatically inserted into the code generated. The code will have method bodies with method invocations and *assertions*. We call such a code a *probably correct code*. Static analysis techniques and tools such as ESC-Java [7] can be used for verification of correctness of the code against the design model.

The Construction Programmer can now work on the generated code with method invocations and assertions, and produce executable code. However, the assertions should not be removed and thus the result should be code with assertions. Testing and static analysis again can be carried out with the aid of tools such as ESC-Java. If the assertions are written in Spec# assertion commands and the Construction Programmer code the program in Spec#, the executable code could be a Spec# program. In this case, the Spec# compiler takes care of the static analysis. We think it would be a significant advantage for Spec# to be realistically useful, as it is not feasible for a programmer to code the assertion commands correctly. The assertions should be generated or carried from the design models.

An important advantage of our proposed method is that these assertions would be already included in the code generated by the Construction Manager from the model, and the Construction Programmer would be bound to follow and aim to assure these assertions.

Before submitting his work to the Construction Manager, a Construction Programmer should provide proof that his work is correct - *Quality Assurance Manager*, a new role to be introduced into MasterCraft, should run a static analysis tool or a run-time monitoring tool (in a test environment) to verify that all the assertions will be satisfied.

3.5. Summary

This section has demonstrated that there is indeed room for further development based on formal methods.

We have found the following main areas:

1. We suggest to introduce an Architectural Model of the components and their dependencies using the new features of UML 2.0. This kind of model can be annotated with *contracts* and thus be checked for interaction consistency and compatibility between required and provided interfaces.
2. For Analysis Modelling, we recommend further support by checking consistency among conceptual class diagram, use cases, sequence diagrams, and state or activity diagrams. Some transformations can also be implemented, such that one can generate rapid prototypes for system validation, improved detailed designs or even use them as implementations.
3. For Design Modelling, we suggest to implement verified correctness preserving transformations in QVT to support stepwise and incremental design. We also think more assertions and checks can be automatically generated for the added classes and methods. This will essentially reuse the techniques proposed for the analysis level.
4. The implementation level can use the assertions generated from the added design and analysis documentation to improve coverage in unit tests. This may also facilitate the use of static analysis and run-time monitoring tools.
5. The application of component models created in different stages in the development would be useful for handling problems like consistency between versions of a component, substitutability of components, composition and assembly of components.

It is encouraging to observe that the disciplined process which is realized by MasterCraft, fits well with the concepts of rigorous development methods that have matured over the last decades. It would demonstrate a much desired link between mature software processes and rigorous development methods.

4. Conclusion and Discussion

We have analyzed the software development process in a commercially successful tool (MasterCraft [29]) and identified where formal methods support can be “plugged” into the tool to make software development more efficient. Already for the initial architectural decisions, a well defined formalism such as rCOS [13, 4] has the means to specify the application’s components;

a precise component specification would not only help in assuring correct component interaction, but also lead to less iterations in the analysis and design modeling.

In the analysis and design modeling stages, refinement rules may assist in creating models *correct by design*. The refinement rules would be implemented as transformations; in addition to the built-in transformations, there would also be support for user-defined transformations. Here, a practical applicability limitation is that the author of the transformations would be required to provide a proof that the added transformations are correct (with respect to either refinement or preservation of desired properties). It may not be always possible or practical to construct the design model solely through a sequence of correct transformations; in such a case, model-checking or other analysis tools have to be used on the resulting model to verify its desired properties. In addition, certain transformations may require side-conditions to be satisfied; these have to be verified either also through model checking, or through static analysis of the final code, or even through run-time monitoring.

Based on the formal semantics of the behavioral diagrams, the code generation framework can be extended to generate templates for *verifiable code* for the method bodies, with assertions (in the style of Spec#) both guiding the programmer in writing the code, and aiding a static analysis tool or a test conductor in verifying that the assertions will be specified.

The proposed project is obviously challenging. Yet, in general, the semantics for state diagrams and sequence diagrams makes it feasible. For instance:

1. With the QVT engine that is being developed at TRDDC, we can program the refactoring rules and design patterns proved in rCOS.
2. Automatic generation of executable code is challenging, however, with the semantics of state diagrams, sequence diagrams and textual specifications, it is possible to generate code with control structures, method invocations, assertions, and class invariants.
3. The most difficult problem is the scaling up of formal method tools. We hope that the separation of concerns in the model of components will help.

In our further work, we plan to integrate into MasterCraft some of the most promising of these, including selected back-end tools, e.g., AutoPA1.0 and other model checking, theorem proving, and static analysis tools. Examples that such a tool integration is feasible can be already found, e.g., in the Evidential Tool Bus [28]. The integration should however be done “loosely”

in order to give the developer the freedom in deciding on the level of formalism employed depending on the application.

References

- [1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, LNCS 3362*, pages 49–69. Springer, 2005.
- [2] G.W. Bond, *et al.* Experience with Component-Based Development of a Telecommunication Service. In G.T. Heineman, *et al.*, editor, *Component-Based Software Engineering, 8th International Symposium, CBSE 2005, LNCS 3489*, St. Louis, MO, USA, 2005. Springer.
- [3] A. Cavalcanti, A. Sampaio, and J. Woodcock. A Refinement Strategy for Circus. *Formal Asp. Comput.*, 15(2-3):146–181, 2003.
- [4] X. Chen, J. He, Z. Liu, and N. Zhan. Component-Based Programming. Technical Report 350, UNU-IIST, P.O. Box 3058, Macao SAR, China, November 2006. Accepted by IPM International Symposium on Fundamentals of Software Engineering, FSEN’07.
- [5] X. Chen, Z. Liu, and V. Mencl. Separation of Concerns and Consistent Integration in Requirements Modelling. In *Proc. of 33rd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 07), LNCS 4362*. Springer, 2007.
- [6] V. Damm. Offering Formal Verification Capabilities for Industry Standard Case Tools: Challenges and Results. In *3rd IEEE International Conference on Formal Engineering Methods, ICFEM 2004*, York, England, UK, 2000. IEEE Computer Society.
- [7] C. Flanagan, *et al.* Extended Static Checking for Java. In *Pro. PLDI’ 2002*, 2002.
- [8] M. Fowler, *et al.* *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [9] E. Gamma, *et al.* *Design Patterns*. Addison-Wesley, 1995.
- [10] Object Management Group. MOF QVT Final Adopted Specification. <http://www.omg.org/docs/ptc/05-11-01.pdf>, ptc/05-11-01, 2005.
- [11] Object Management Group. Unified Modeling Language: Superstructure, version 2.0, final adopted specification. <http://www.omg.org/uml/>, formal/05-07-04, 2005.
- [12] J.V. Guillen-Scholten, F. Arbab, F.S. de Boer, and M.M. Bonsangue. A Component Coordination Model Based on Mobile Channels. *Fundamenta Informaticae*, To appear, 2006.
- [13] J. He, X. Li, and Z. Liu. rCOS: A refinement calculus for object systems. *Theoretical Computer Science*, 365(1-2):109-142, 2006.
- [14] D. Herzberg and M. Broy. Modeling layered distributed communication systems. *Formal Asp. Comput.*, 17(1):1–18, 2005.
- [15] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
- [16] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11:256–290, 2002.
- [17] P. Kruchten. *The Rational Unified Process – An Introduction*. Addison-Wesley, 2000.
- [18] C. Larman. *Applying UML and Patterns*. Prentice-Hall Intl., 3rd edition, 2005.
- [19] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical Report 98-06-rev29, Department of Computer Science, Iowa State University, USA., January 2006.
- [20] Z. Liu, J. He, X. Li, and Y. Chen. A relational model for formal requirements analysis in UML. In J.S. Dong and J. Woodcock, editors, *Formal Methods and Software Engineering, ICFEM03, LNCS 2885*, pages 641–664. Springer, 2003.
- [21] Z. Liu, V. Mencl, A. P. Ravn, and L. Yang. Harnessing Theories for Tool Support. Technical Report 343, UNU-IIST, P.O. Box 3058, Macao SAR, China, August 2006.
- [22] Q. Long, J. He, and Z. Liu. Refactoring and Pattern-directed Refactoring: A Formal Perspective. Technical Report 318, UNU/IIST, P.O. Box 3058, Macau, 2005.
- [23] Q. Long, Z. Liu, X. Li, and J. He. Consistent code generation from UML models. In *Pro. of Australian Software Engineering Conference (ASWEC’2005)*, Brisbane, Australia, 2005. IEEE Computer Society.
- [24] Q. Long, Z. Qiu, Z. Liu, L. Shao, and J. He. POST: a case study for an incremental development in rCOS. In *Proc. 2nd International Colloquium on Theoretical Aspects of Computing (ICTAC 2005), LNCS 3722*, pages 485–500. Springer, 2005.
- [25] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/Hol: A Proof Assistant for Higher-Order Logic*, volume 2283 of LNCS. Springer, 2002.
- [26] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Pro. 11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1991.
- [27] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
- [28] J.M. Rushby. An Evidential Tool Bus. In *Pro. ICFEM 2005*, pages 36–36, 2005.
- [29] Tata Consultancy Services. MasterCraft. <http://www.tata-mastercraft.com/>.
- [30] Y. Wei, X. Li, Z. Liu, and J. He. Automatic Transformation from Requirements models to Executable Prototypes. Technical Report 329, UNU-IIST, P.O. Box 3058, Macao SAR, China, October 2005.
- [31] L. Yang, V. Mencl, V. Stolz, and Z. Liu. Automating Correctness Preserving Model-to-Model Transformation in MDA. In *Draft Proc. of the 1st Asian Working Conference on Verified Software, AWCVS’06*, UNU-IIST Report No. 348, 2006.