
Harnessing Theories for Tool Support

Zhiming Liu, Vladimir Mencl, Anders P. Ravn,
and Lu Yang

August 2006

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: <http://www.iist.unu.edu>, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director

P.O. Box 3058
Macao

Harnessing Theories for Tool Support

Zhiming Liu, Vladimir Mencl, Anders P. Ravn,
and Lu Yang

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 how to give conditions such that they will actually be used.

Zhiming Liu is a Research Fellow at UNU-IIST. His research interests include theory of computing systems, emphasising sound methods for specification, verification and refinement of fault-tolerant, realtime and concurrent systems, and formal techniques for OO development. His teaching interests are Communication, Concurrent and Distributed Programming, Internet Security, Software Engineering, Formal specification and Design of Computer Systems. His email address is Z.Liu@iist.unu.edu

Vladimir Mencl is a Visiting Researcher at UNU-IIST and a Researcher at the Charles University in Prague, Czech Republic, where he is a member of the Distributed Systems Research Group, <http://nenya.ms.mff.cuni.cz/>. His research interests are centered around component-based systems, and include formal methods for specifying behavior of software components, as well as the structural aspects of component models. His email address is mencl@iist.unu.edu.

Anders P. Ravn is a Professor of Computing at the Department of Computer Science, Aalborg University. He works in the Distributed Systems and Semantics Unit and also associated with BRICS, CISS, and Center for Agricultural Technology centres within the Faculty of Engineering and Science. His broad interest is methods for development of embedded systems, in particular the engineering of software for such systems. He believes that software engineering, like other more mature engineering disciplines, shall have a foundation in mathematical theories and then explain by prescriptions or methods, how the theories are applied to solve classes of problems. His email address is apr@cs.aau.dk.

Yang Lu is a fellow of UNU-IIST from Nanjing University, Nanjing, China, where she is a Ph.D. student. Her research interests include formal methods, web services, and model driven architecture. Her email address is yanglu@iist.unu.edu.

Contents

1	Introduction	1
2	Tool supported software development - MasterCraft	2
2.1	Concepts in MasterCraft	3
2.2	Roles in software development	4
3	Desirable Formal Support	10
3.1	Component modelling	10
3.2	Analysis modelling	12
3.3	Design modelling & model transformations	12
3.4	Construction & code generation	13
3.5	Summary	14
4	Conclusion and discussion	15

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 assembling code, even when they are supported by extensive libraries with many utilities. There is a need for tools that support the development process, such that the developers can focus on modelling the application and let 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) [15], 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 nice term was coined in the telecommunications industry [3] 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 [5, 1, 4, 8, 23, 26]. 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 [10, 9]. 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 [43]. 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 [30, 18, 7]¹ and is based on the UTP framework [22], 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 CSP-OZ in [39] and Circus [6] 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 [21] and Reo [16] 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 Tool supported software development - MasterCraft

MasterCraft is developed by TRDCC [43] 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 (or informally called *artifacts*) 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 first briefly introducing the concepts and artifacts considered, and afterwards by elaborating on the roles and their responsibilities. The concepts and artifacts, as well as their relations, are shown by the UML class diagram in Fig. 1, while the responsibilities and activities of the roles are illustrated by the sequence diagrams in Fig. 2 and Fig. 6. We give the explanations in the following subsections. In Section 3, we will discuss how to give formal semantics to these concepts and artifacts, and suggest formal method tools to support the roles in their activities.

¹The technical report [7] contains the most recent development in rCOS.

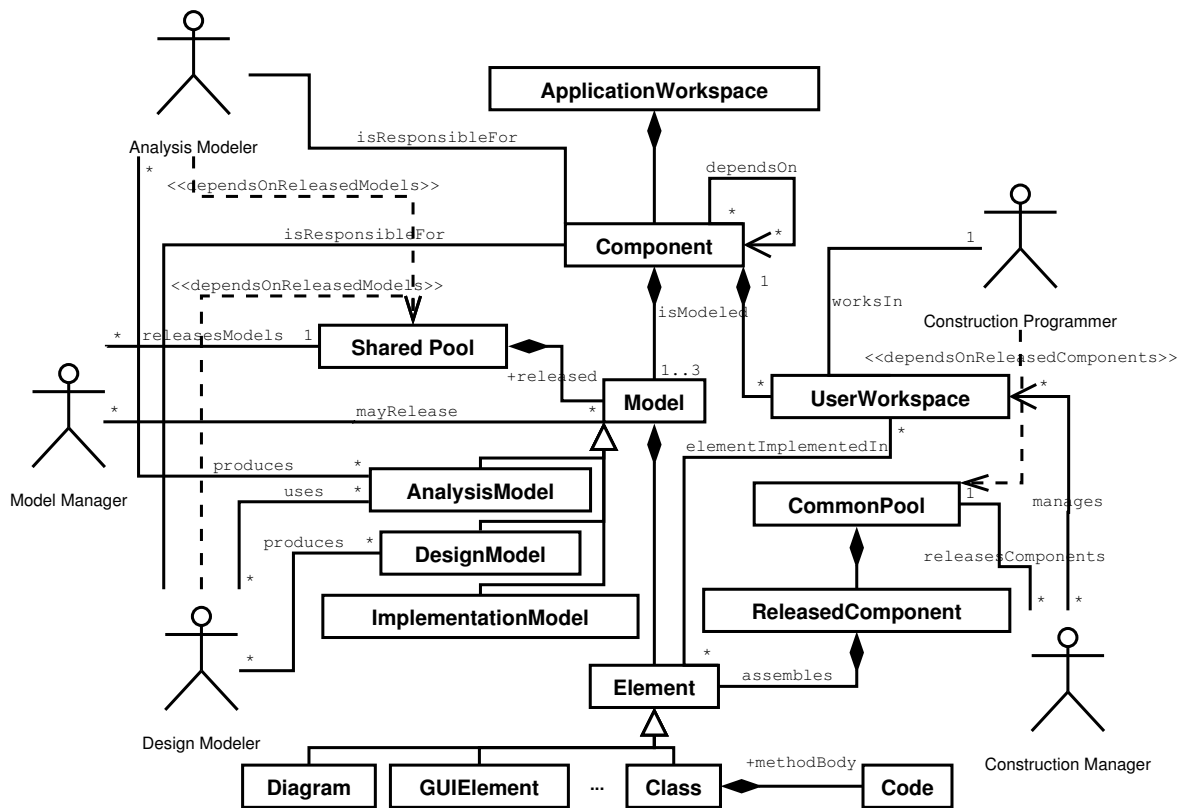


Figure 1: MasterCraft: class diagram of process-oriented concepts.

2.1 Concepts in MasterCraft

MasterCraft introduces a body of concepts and a hierarchy of artifact repositories, designed to support team collaboration on development of the models and code. Figure 1 shows the relations among these concepts as a class diagram. At the top-level of component repositories is the *application workspace*, representing the whole modelling and development space of an application. The application workspace is further partitioned into *components*. Different from conventional component-based software development (CBSD), MasterCraft focuses on component development and is oriented towards organizing the development activities. Nevertheless, a component is characterized by its interface (consisting however only of the component's provided operations) and its dependencies on other components.

As analysis and design models are created in the individual components of the application workspace, stable versions of these models can be released into the *shared pool*. This allows developers of other components, depending on the components already released, to use stable versions of the models. In order to preserve consistency, once the model has been released, it is “frozen” and any subsequent change starts a new modelling cycle; this is also reflected by a change in the version identifier of the new model.

The models in MasterCraft are created as instances of a metamodel based on UML. Besides the modelling constructs already available in UML, MasterCraft introduces a few technology-oriented concepts, such as

database queries (eventually translated into classes), and also several concepts for modelling the graphical user interface (GUI) of the application. The GUI interacts with the application by invoking operations provided by a classes. To assist with managing consistency of data type definitions, MasterCraft introduces a single component **domains** that defines the simple data types, that is, the primitive types and types constructed from them. These simple data types are called *domain types*.

In parallel with the shared pool, the *common pool* is a repository of code artifacts, where stable implementations of components are released. Such stable releases of component implementations can be used by developers of dependent components.

While a single programmer (a Construction Programmer, as the role will be named later) works on the assigned tasks for a component (such as classes to be coded), the development takes place in a separate development area called *user workspace*. Only after the tasks are completed (including unit testing), the code is committed into the application workspace.

2.2 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*. Different roles give different rights to access different artifacts of the project. Their activities are illustrated in the sequence diagrams in Figs. 2 and 6.

Administrative tasks.

At the very beginning of the development, the *Administrator* is responsible for creating user accounts and components, and assigning roles to project participants for acting on the components they are involved in. As the development of the application progresses, if a version control system is in use, the *Version Manager* may store snapshots of the whole application workspace (the models and code it contains) in the version control system repository, and if needed, restore them as a separate application workspace for parallel development.

Example.

We will illustrate software development in MasterCraft on the case study of a simple Point-Of-Sales Terminal (POST) System originally proposed by Larman [25]. 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. Naturally, the operation of the system is subject to a number of constraints, such as “the total of a sale must be equal to the sum of the cost of its item” or “when a sale is completed and paid, the total of a sale must be equal to the book value of a payment, i.e., amount paid minus change returned”.

The administrator starts by creating the components identified, **PaymentManagement**, **SalesManagement**, and **InventoryManagement** in this example. Next, the administrator creates user accounts, let’s say *Alice* and *Brian*, and assigns them roles. In this case, Alice may become both Analysis Modeler and Design

Modeler for InventoryManagement and SalesManagement, and Brian may be granted these roles for the PaymentManagement component. Furthermore, we have *Martin* who is assigned the global role of Model Manager.

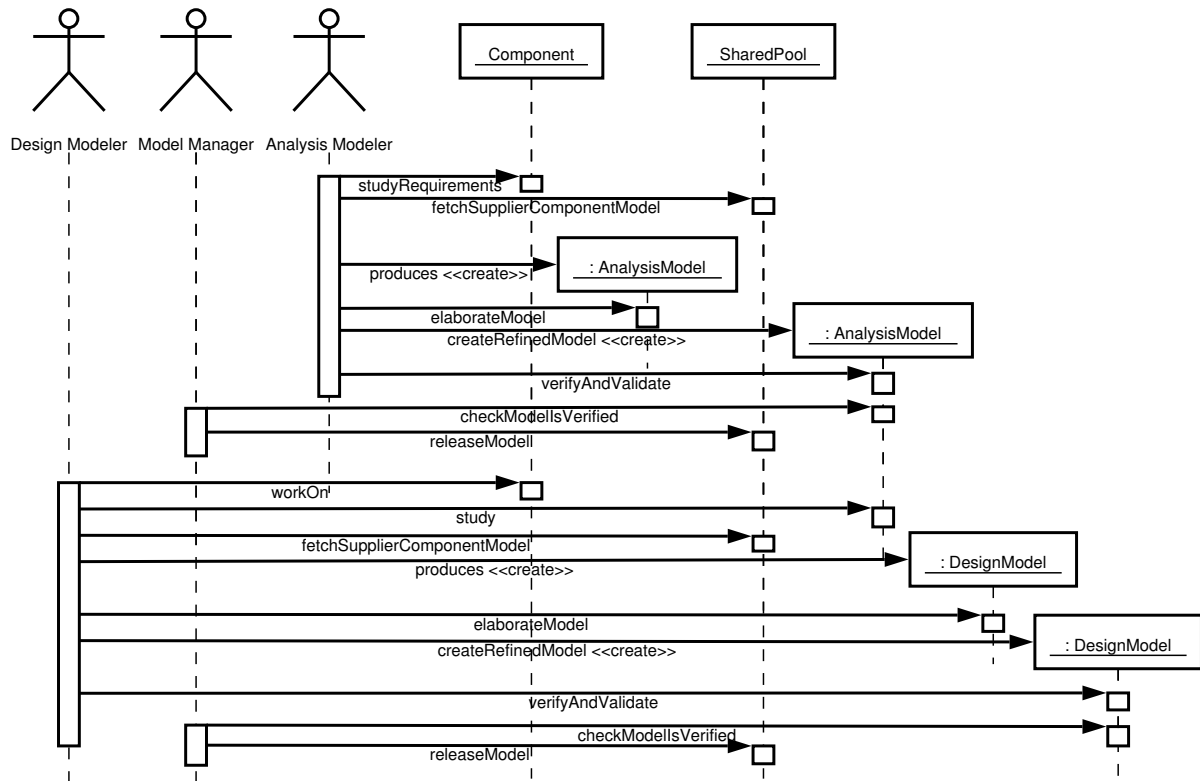


Figure 2: MasterCraft: sequence diagram of modelling tasks.

Modelling tasks.

An *Analysis Modeler* starts work on a component by studying its textual requirements, kept outside of MasterCraft. Based on the textual requirements, the Analysis Modeler creates an analysis model. An analysis model consists of conceptual class diagrams, use case diagrams, and behavioral models of the use cases, captured as sequence, collaboration, activity and state machine diagrams. The Analysis Modeler may iterate over this model, creating a new refined model based on the original analysis model. The Analysis Modeler can declare a dependency on another component and, if the component depends on other components, the Analysis Modeler first fetches the models of these *supplier components* from the shared pool. Upon completing the model, the Analysis Modeler is responsible for verifying that the model is consistent, and validating that it realizes the requirements.

The *Model Manager* can afterwards release the model into the shared pool, making it available for Analysis Modelers working on components depending on this component. After being released into the shared pool, the model in the application workspace is frozen, and any additional changes would start a new modelling cycle. Before releasing the model into the shared pool, the model manager has to ensure that the Analysis Modeler has validated the model.

A *Design Modeler* refines the analysis model of a component into a design model. The conceptual classes from the analysis model are refined into design classes. This includes modelling operations provided by the classes (so far, the operations are modelled only at the syntactic level with their signatures). The Design Modeler decides on which classes should be persistent, and defines database mapping and primary keys for these classes. The Design Modeler also plans database queries to access the persistent data, and models the application's GUI (as Windows, Controls and WinClasses), and the interaction of the GUI with the application in terms of operation calls. Further, the Design Modeler defines the *component interface* in terms of class operations and queries provided, and may declare additional dependencies on other components. Note that before commencing the work on the design model, the Design Modeler needs to fetch models of the supplier components from the shared pool. Just as the Analysis Modeler, the Design Modeler may also iterate over the design model, refining it into a new version. Upon completing the work on the design model, the Design Modeler is responsible for verifying its consistency, and validating it with respect to the analysis model.

The Model Manager has the same responsibility for design models as for analysis models — and after checking that the design modeler did the verifications and validations required, the Model Manager may release the design model into the shared pool. We illustrate a typical flow of the modelling tasks in Fig. 2.

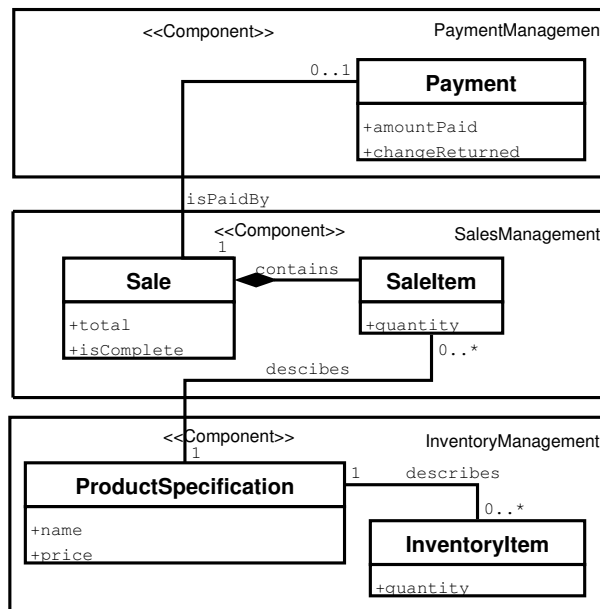


Figure 3: Point-of-sale case study analysis class diagram

Example.

In our POST case study, Alice starts with analysis modelling of the `InventoryManagement` component, and creates there conceptual classes `ProductSpecification` and `InventoryItem`. In the iterative work on the model, the classes are later linked with an association `describes`. After validating the analysis model, Alice continues with the component `SalesManagement`, and identifies that this component depends on the components `InventoryManagement` and `PaymentManagement`; the latter has to be first completed by Brian. After Brian creates the analysis model (containing a single conceptual class `Payment`) and validates it, Martin releases models of both the already completed components into the shared pool.

Alice can then fetch the released models, and proceed with work on the `SalesManagement` component. Figure 3 shows the classes created in the POST 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. 4 (a) and state machine shown in Fig. 4 (b). After the model of this component is created and validated, all the components proceeds into the design stage. The design model created based on the analysis model adds new elements needed for the design — the operations provided by classes are now modelled, navigability of associations is decided, and additional attributes and classes may be added if needed. We illustrate this on the *design class diagram* shown in Fig. 5. In addition, design-level behavioral diagrams now model the interaction among the classes; we illustrate this on the *design sequence diagram* shown in Fig. 4 (c). Further, the GUI may be modelled for each of the components. Eventually, the design models are validated and released into the shared pool.

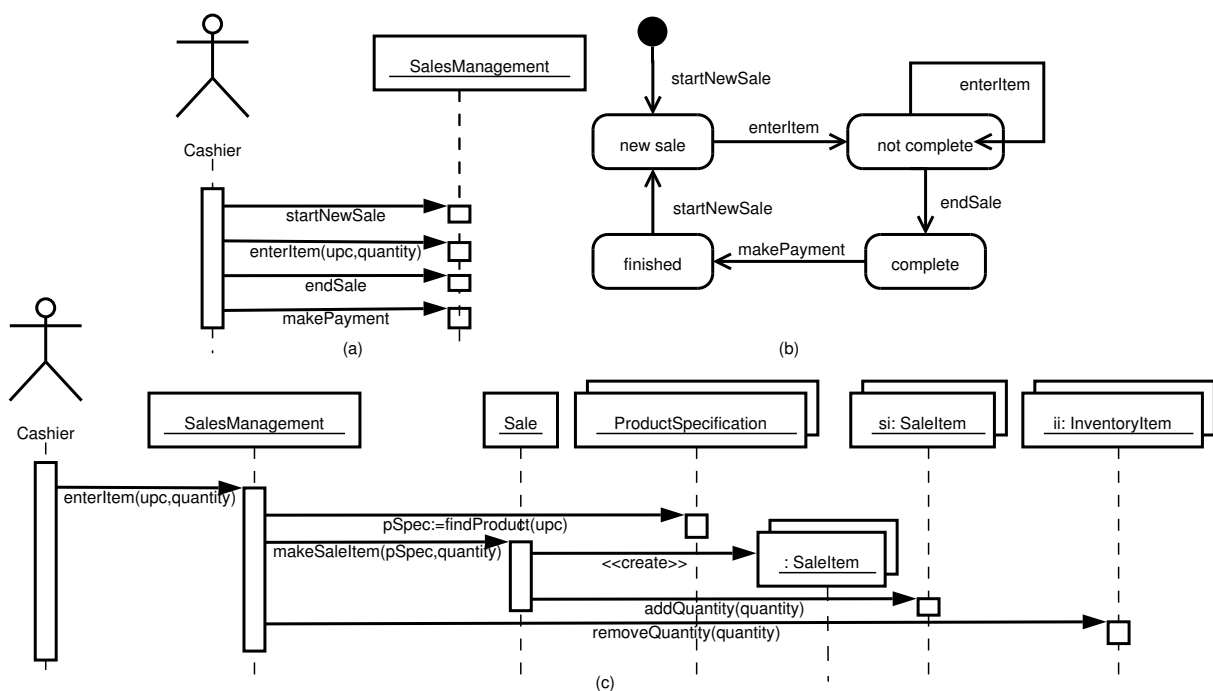


Figure 4: Point-of-sale case study analysis sequence diagram (a), analysis object state diagram (b) and design sequence diagram (c)

Construction tasks.

The *Construction Manager* is the key role responsible for construction tasks. The Construction Manager starts by exporting the design model of a Component into an external representation, and invokes code generation tools, which generate code templates for all the classes. The code template of a class contains for each attribute of the class its declaration and accessor methods. For persistent classes, the code template also contains database interaction methods for transferring the state of the class between its attributes and a relational database. Further, the code template contains declarations of all the operations declared for the class. However, implementations of the operations defined in the design model are missing. Subsequently, the Construction Manager assigns coding of these operations (as well as coding of Queries) to *Construction Programmers* by *defining* a User Workspace for each selected Construction

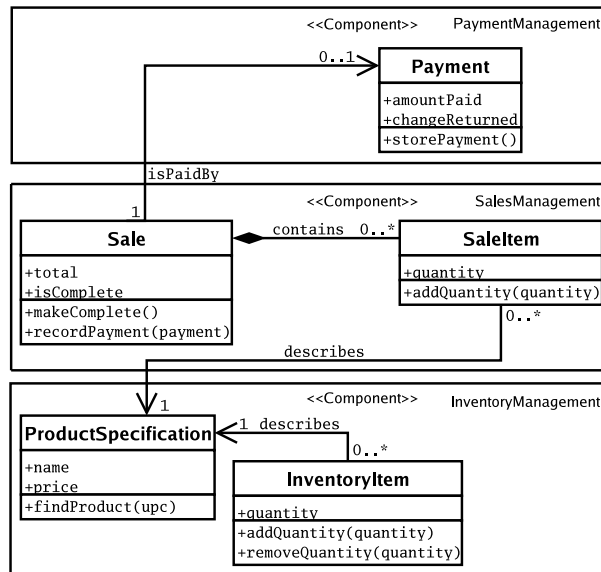


Figure 5: Point-of-sale case study design class diagram

Programmer. A Construction Programmer starts work by *fetching* the workspace. After coding and unit testing the assigned operations and Queries, the Construction Programmer *builds* the workspace. Finally, the Construction Manager accepts the code by *synchronizing* the workspace, and eventually *dissolves* the workspace. After receiving code for all the tasks assigned to different Construction Programmers, the Construction Manager integrates the code together. After integration testing of the code of the Component, the Construction Manager releases the compiled binary code of the component into the common pool, making it available for development of other, dependent components. We illustrate the construction tasks in the sequence diagram in Fig. 6.

Example.

To start construction work on the POST system, the Administrator now assigns Alice also the role of Construction Manager, and assigns two Construction Programmers to the project, *Chris* to implement the `PaymentManagement` component, and *David* for `InventoryManagement` and `SalesManagement`. Alice as the Construction Manager assigns David to code all the methods of the `Payment` class, by creating him a user workspace, and also assigns Chris to code methods of the classes `InventoryItem` and `ProductSpecification` in `InventoryManagement`. They both fetch their workspaces, code and unit test their methods, and submit their work by building the workspaces. After Alice synchronizes their workspaces, she creates binary releases of both the already completed components, and releases them into the common pool. Now, she can create for David a new user workspace, assigning him to code the methods of the `Sale` and `SaleItem` classes of the `SalesManagement` component. After he finishes, Alice builds and releases the component into the common pool, and the construction work is complete.

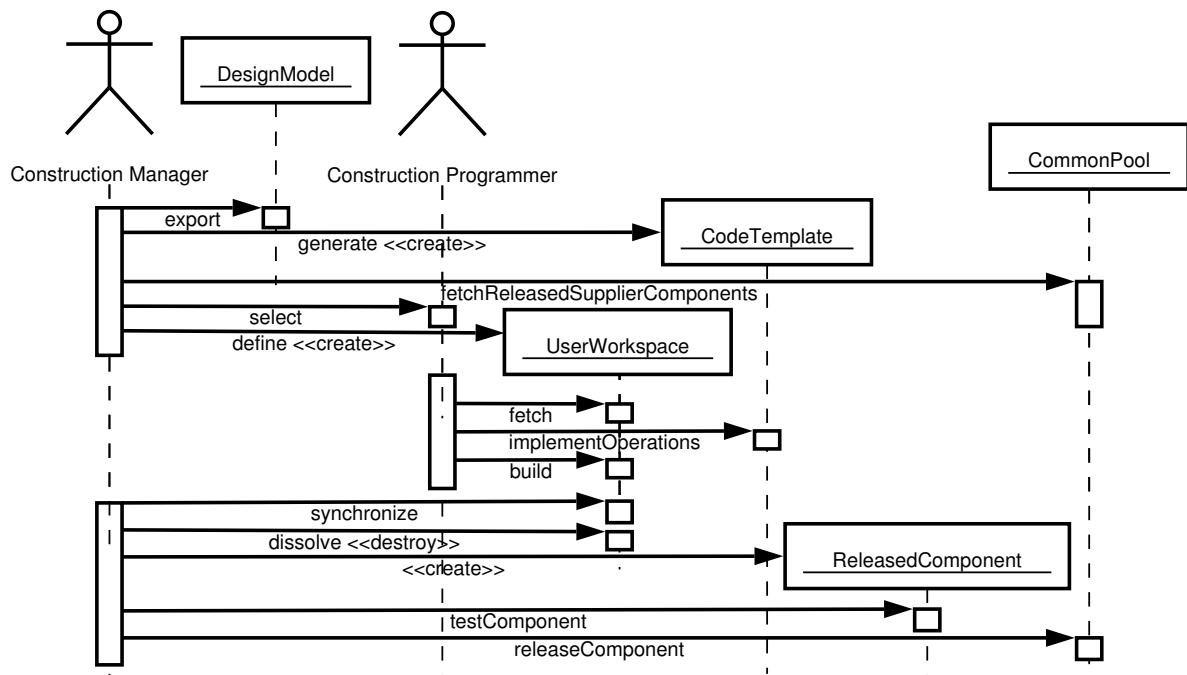


Figure 6: MasterCraft: sequence diagram of construction tasks.

Summary.

Throughout this section, we have illustrated how the roles the developers take cooperate in the software development process driven by MasterCraft. We now summarize responsibilities of the roles in terms of the input artifacts they take and output artifacts they produce. At the very beginning, the Administrator takes the architectural description of the application, and creates Components, user accounts, and assigns project participants the roles they will have with the Components. Afterwards, for each component, the Analysis Modeler takes its textual requirement, and produces an analysis model. Subsequently, the Design Modeler takes the analysis model, and produces a design model. A Model Manager releases stable versions of both analysis and design models of a component into the shared pool; from there, the models are available to Analysis and Design Modelers for work on components depending on the component already finished. Further, in iterative development, Analysis and Design Modelers may also use as input earlier versions of the models they work on.

A Construction Manager takes a completed design model, generates code template, and creates user workspaces for Construction Programmers with coding tasks (method bodies to be implemented). The Construction Programmers develop code inside their workspaces; afterwards, the Construction Manager integrates the code developed in user workspaces, and creates a binary release of the component, later released into the common pool. From there, the code of a component is accessible to Construction Programmers for development of components depending on the already implemented component. Anytime during the development, a version manager may take all the models and code stored in the application workspace, and store them in a version control repository, and may later restore such a snapshot as a separate application workspace.

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 POST 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 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.

Thus there is ample room for improvement based on research on semantics of object and component systems [19, 20, 30, 17, 18, 7], together with their applications in formalizing constructs of UML [27, 32, 31, 29, 33, 36, 37]. 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 in 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 [15]. 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), to describe features of states and method signatures [30], and at the semantic level (Contracts of Interfaces) [17] including aspects of *functionality* [30], *interactions* (Protocols) and *behaviors* [7]. A component diagram for POST² is shown in Fig. 7.

In the framework of rCOS [18, 7, 30], 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

²This is a rather simplified case study. For example, the component *PaymentManagement* can be extended to *FinanceManagement*. In that case, interfaces between *FinanceManagement* and *InventoryManagement* are required.

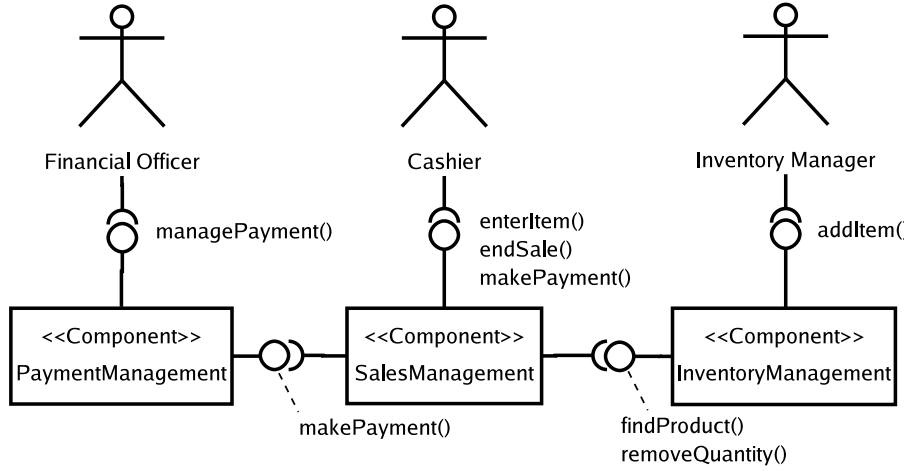


Figure 7: Point-of-sale case study component diagram

defines the functionality by specifying a precondition and a post condition for each operation, and by specifying a protocol as a set of finite traces (sequences) of operations.

For example, assume that component `PaymentManagement` has two state variables `amount` and `change` of a predefined domain type `Currency`. Operation `makePayment(a:Currency,s:Currency)` in the interface provided to component `SalesManagement` takes as inputs the amount `a` received from the customer and the sale's total `s` from the sale object, and sets the fields `amount` to `a`, and `change` to `amount - s`. This can be specified in its precondition ($a \geq s$) and postcondition $amount' = a \wedge change' = (a - s)$ in the form of a UTP design

$$(a \geq s) \vdash amount' = a \wedge change' = (a - s)$$

The protocol is supplemented with guard conditions for each operation, such that it together with the functionalities defines a set of CSP *failures* and *divergences* [41], though the language of CSP is not employed.

A contract defines an automaton or a CSP process (with method invocations and returns as the events), and thus formal analysis tools, such as SAL or FDR [41] can support analysis and verification of properties, in particular intra-component interactions. 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 provided and required interfaces. For this, we can check the interaction compatibility and functional compatibility separately, possibly with different tools. 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 proof checkers [40, 38].

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 application of component models created in different stages in the development would be 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 [31].

Simple use cases, such *MakePayment* in the POST example, can be defined as a one step operation by a precondition and post condition. An invariant can be specified in OCL and checked by MasterCraft [34]. In general, a use case, such as the *BuyItems* use case of POST, 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. We have made progress in this direction in [28, 45].

In [31], we give formal definition of requirements analysis models and the consistency relation between the class models 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 [28, 45]. Based on these, an initial version of a tool, called AutoPA1.0, is developed for automatic generation of executable prototypes of analysis models [29]. 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 [36]. 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 Design modelling & model transformations

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 a *design class diagram* (or packages of class diagrams), *object sequence diagrams* and *object state diagrams*. The design class diagram gives detailed declarations of attributes and method signatures of each class, and specifies the inheritance,

association and dependency relations among classes (cf. Figure 5). 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. Figure 4 (c)). An object state diagram defines the behavior of an (important) object. In MasterCraft, the class 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) [24], is usually adopted. In such a process, each subsequent version should be produced from its preceding version by applying a transformation that preserves already specified properties. In fact, the sequence diagram in Fig. 4 (c) is created by a number of applications of the design patterns for object responsibility assignment, the *Expert Pattern* [25] in particular. The design class diagram in Fig. 5 is constructed from the analysis class diagram in Fig. 3 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 [26] and Alloy [23]. 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 refinement rules and design patterns [13] and refactorings [12] can be programmed in a transformation language such as QVT [14]. For the patterns *expert*, *class decomposition*³, and *attribute encapsulation*, a proof of their correctness with respect to refinement has already been given in [18] and applied to the development of the POST system [37]. Refactoring rules are proved in rCOS in [35] and applied in the case study POST in [37]. Extending MasterCraft with implementations of these design patterns and refinement rules will enhance the support provided to the Design Modeler in producing good designs.

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.

With semantics of sequence diagrams in [28] and the algorithms in [36], 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). Code generated in [36] for a simple example is given below just to show the promise.

³More widely known as High Cohesion and Low Coupling Patterns.

```

class A {          | class C-handler{  | class B1{
  B refB1;        |   C element;      |   B2 refB2;
  C refC;         |   Vector<C> vector; |   meth_1(){
  meth_0(){       |   }               |   refB2.meth_9();
    refB1.meth_1(); |               |   }
    command_3;    |   class C{        |   }
    (g)* {        |   A refA;         |   }
    refC.meth_4(); |   meth_4(){       |   class B2{
  }               |   refA.meth_5();  |   meth_9(){
}                 |   }               |   SD_11();
meth_5(){        |   }               |   }
}                 |                   |   }

```

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 [2] 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 coded 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 ESCJava [11] 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 ESCJava. 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 would be 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 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 and design patterns 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 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 [43]) and identified where support of formal methods 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 [18, 7] has the means to specify the application’s components; such a precise component specification would not only help in assuring correct component interaction, but should 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 refinement 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 [42] and the Evolution and Validation Environment [44]. The integration should however be done “loosely” in order to give the developer the freedom in deciding the level of formalism employed depending on the application.

Acknowledgments.

We would like to thank Mathai Joseph and R. Venky from TRDDC for their constant help in our understanding of MasterCraft and their useful comments on early versions of the paper. Xiaoshan Li at the University of Macao has been working with us on related problems and attended many of our discussions on this paper.

References

- [1] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003.
- [2] 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.
- [3] G.W. Bond, E. Cheung, H. Goguen, K.J. Hanson, D. Henderson, G.M. Karam, K.H. Purdy, T.M. Smith, and P. Zave. Experience with component-based development of a telecommunication service. In G.T. Heineman, I. Crnkovic, H.W. Schmidt, J.A. Stafford, C.A. Szyperski, and K.C. Wallnau, editors, *Component-Based Software Engineering, 8th International Symposium, CBSE 2005, LNCS 3489*, St. Louis, MO, USA, 2005. Springer.
- [4] M. Bozga, S. Graf, Illeana, Iulian Ober, and J. Sifakis. The if toolset. In M. Bernardo and F. Corradini, editors, *International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, LNCS 3185*, Bertinoro, Italy, 2004. Springer.
- [5] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement*. Springer, 2001.

- [6] A. Cavalcanti, A. Sampaio, and J. Woodcock. A refinement strategy for circus. *Formal Asp. Comput.*, 15(2-3):146–181, 2003.
- [7] X. Chen, J. He, and Z. Liu. Component coordination in rCOS. Technical Report 335, UNU-IIST, P.O. Box 3058, Macao SAR, China, May 2006.
- [8] A. Childs, J. Greenwald, V.P. Ranganath, X. Deng, M.B. Dwyer, J. Hatcliff, G. Jung, P. Shanti, and G. Singh. Cadena: An integrated development environment for analysis, synthesis, and verification of component-based systems. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, LNCS 2984*, Barcelona, Spain, 2004. Springer.
- [9] P. Combes, D. Harel, and H. Kugler. Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. In *Automated Technology for Verification and Analysis, Third International Symposium, ATVA 2005, LNCS 3707*, Taiwan, 2004. Springer.
- [10] 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.
- [11] C. Flanagan, K. Rustan M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended Static Checking for Java. In *Pro. PLDI' 2002*, 2002.
- [12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [13] E. Gamma, *et al.* *Design Patterns*. Addison-Wesley, 1995.
- [14] Object Management Group. MOF QVT Final Adopted Specification, ptc/05-11-01. <http://www.omg.org/docs/ptc/05-11-01.pdf>, 2005.
- [15] Object Management Group. Unified Modeling Language: Superstructure, version 2.0, final adopted specification. <http://www.omg.org/uml/>, formal/05-07-04, 2005.
- [16] J.V. Guillen-Scholten, F.S. de Boer F. Arbab, and M.M. Bonsangue. A component coordination model based on mobile channels. *Fundamenta Informaticae*, To appear, 2006.
- [17] J. He, X. Li, and Z. Liu. Component-based software engineering. In *Pro. ICTAC'2005, Lecture Notes in Computer Science 3722*. Springer, 2005.
- [18] J. He, X. Li, and Z. Liu. rCOS: A refinement calculus for object systems. *Theoretical Computer Science*, 2006. accepted, in press, doi:10.1016/j.tcs.2006.07.034, also available as Technical Report 322, UNU-IIST, P.O. Box 3058, Macao SAR China. <http://www.iist.unu.edu/>.
- [19] J. He, Z. Liu, and X. Li. A component calculus. In H.D. Van and Z. Liu, editors, *Proc. Of FME03 Workshop on Formal Aspects of Component Software (FACS03), UNU/IIST Technical Report 284, UNU/IIST, P.O. Box 3058, Macao*, Pisa, Italy, 2003.
- [20] J. He, Z. Liu, X. Li, and S. Qin. A relational model for object-oriented designs. In *Pro. APLAS'2004, Lecture Notes in Computer Science*, Taiwan, 2004. Springer.
- [21] D. Herzberg and M. Broy. Modeling layered distributed communication systems. *Formal Asp. Comput.*, 17(1):1–18, 2005.
- [22] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

- [23] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11:256–290, 2002.
- [24] P. Kruchten. *The Rational Unified Process – An Introduction*. Addison-Wesley, 2000.
- [25] C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 2001.
- [26] 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.
- [27] X. Li, Z. Liu, and J. He. Formal and use-case driven requirement analysis in UML. In *COMPSAC01*, pages 215–224, Illinois, USA, October 2001. IEEE Computer Society.
- [28] X. Li, Z. Liu, and J. He. A formal semantics of UML sequence diagrams. In *Pro. of Australian Software Engineering Conference (ASWEC'2004)*, Melbourne, Australia, 2004. IEEE Computer Society.
- [29] X. Li, Z. Liu, J. He, and Q. Long. Generating prototypes from a UML model of requirements. In *International Conference on Distributed Computing and Internet Technology (ICDIT2004)*, *Lecture Notes in Computer Science*, Bhubaneswar, India, 2004. Springer.
- [30] Z. Liu, J. He, and X. Li. Contract oriented development of component software. In *Proc. IFIP TCS*, pages 349–366, 2004.
- [31] 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.
- [32] Z. Liu, X. Li, and J. He. Using transition systems to unify UML models. Technical report, Dept. of Maths and Computer Science, the University of Leicester, England., May 2002.
- [33] Z. Liu, X. Li, J. Liu, and J. He. Integrating and refining UML models. Technical Report 295, UNU/IIST, P.O. Box 3058, Macao SAR China, 2004. Presented at UML 2004 Workshop on Consistency Problems in UML-based Software Development, October 10-15, 2004, Lisbon, Portuga.
- [34] Z. Liu and R. Venkatesh. Tools for formal software engineering. presented at the IFIP Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE, <http://vstte.ethz.ch/>), held on 10-13 Oct, 2005, Zurich, October 2005.
- [35] 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. A revised version is submitted for publication.
- [36] 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.
- [37] 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.
- [38] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/Hol: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [39] E-R. Olderog and H. Wehrheim. Specification and inheritance in csp-oz. In *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, LNCS 2853*, Leiden, the Netherlands, 2003. Springer.

-
- [40] 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.
- [41] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
- [42] J.M. Rushby. An Evidential Tool Bus. In *Pro. ICFEM 2005*, pages 36–36, 2005.
- [43] Tata Consultancy Services. Mastercraft. <http://www.tata-mastercraft.com/>.
- [44] J. G. Süß, A. Leicher, H. Weber, and R.-D. Kutsche. Model-centric engineering with the evolution and validation environment. In *Proc. of UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th Intl. conf., San Francisco, CA, USA, Oct. 20-24, LNCS 2863*, pages 31–43. Springer, 2003.
- [45] J. Yang, Q. Long, Z. Liu, and X. Li. A predicative semantic model for integrating uml models. In *Pro. ICTAC'2004, Lecture Notes in Computer Science 3407*, Guiyang, China, 2004. Springer.