Harnessing rCOS for Tool Support —The CoCoME Experience[⋆]

Zhenbang Chen[1], Xiaoshan Li[2], Zhiming Liu[1⋆⋆], Volker Stolz[1], and Lu Yang[1]

[1] United Nations University
Institute for Software Technology (UNU-IIST)
[2] Faculty of Science and Technology, The University of Macau

Abstract. Complexity of software development has to be dealt with by dividing the different aspects and different views of the system and separating different concerns in the design. This implies the need of different modelling notations and tools to support more and more phases of the entire development process. To ensure the correctness of the models produced, 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. This paper reports our experience in the work on the Common Component Modelling Example (CoCoME) and shows where such add-ins will fit. In particular, we show how the formal techniques developed in rCOS can be integrated into a component-based development process, and where it can be integrated in and provide extension to an existing successful commercial tool for adding formally supported checking, transformation and generation modules.

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

1 Introduction

Software engineering is now facing two major challenges on

1. how to handle the huge complexity of the development of a system, and
2. how to ensure the correctness and quality of the software

The complexity of software development is inherent due to many different aspects of the system, including those of static structure, flow of control, interactions and functionality, and different concerns of functionality correctness, concurrency, distribution, mobility, security, timing, and so on. Large software development

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

⋆⋆ I started working on separation and integration of models of different aspects and concerns of systems when I started my study on Master Degree [24] under the supervision of Professor Zhou Chaochen.

requires a large team of people playing different roles and carrying out different activities of design, construction, analysis, verification and validation. The management of the development is complex too.

In *practical software engineering* nowadays, complexity is dealt with by a component-based and model-driven development process [8, 9] where

1. the different aspects and views of the system are described in a UML-like multi-view and multi-notational language, and
2. separation of design and validation of different concerns is supported by design patterns, object-oriented and component-based designs.

However, there are no rigorous unified theories and tools which support specification, verification and validation of the models produced in such a process.

Rigorous verification and validation of a software system requires the application of formal methods. This needs a formal version of the requirements specification, and the establishment of a property to imply that specified requirement holds as long as the assumptions hold. The assumptions are specifications for or constraints on the behavior of environment and system elements. In the past half a century, semantic foundations, formal notations and techniques and tools of verification and validation have been developed, including *testing*, *static analysis*, *model checking*, *formal proof and theorem proving*, and *runtime checking*. They can be classified into the following frameworks:

– event-based models [29, 15] are widely used for specification and verification of interactions, and are supported by model checking and simulation tools [30, 2].
– pre-post conditions and Hoare logic are applied to specifications of functionality and static analysis. These are supported by tools of theorem proving, runtime checking and testing [22, 11, 28].
– state transition systems and temporal logics are popular for specification and verification of dynamic control behaviours. They are supported by model checking tools [17, 21].

However, each framework is researched mostly by a separate community, and most of the research in verification has largely ignored the impact of design methods on feasibility of formal verification. Therefore, the formal techniques and tools are not good with regard to scalability and they are not easy to be integrated into practical design and development processes. The notion of *program refinement* [5] has obvious links to the practical design of programs with the consideration of abstraction and correctness, but the existing refinement calculi are shown to be effective only for small imperative programs. There is a lack of a formal foundation for object-oriented and component-based *model refinement* until the recent work on formal methods of component and object systems [10, 25, 14, 7].

The formalism, rCOS [14, 7], that we have recently developed, is a rather rich and mature formalism that models static and dynamic features for component based systems. It is based on the UTP framework [16], and its accompanying methodology of *separation of concerns* [8], have been applied in a case study of a Point Of Sale terminal within the CoCoME (Common Component Modelling Example) challenge [4]. In this paper, we discuss our experience on how the construction of formal models and their verification and validation can be integrated in a use case driven and component-based development process. In particular, we will show with examples from the CoCoME case studies

1. what the models of the different aspects of the systems are at each stage of the development, including the *requirement elicitation*, *logic design*, *detailed design*, *code generation*,
2. how these models are constructed and derived by application of design patterns that are proved to be a refinement in rCOS, and
3. how verification and validation tasks are identified for the models and what are the effective tools for these tasks.

With regard to model construction and derivation, we focus on the aspects of interactions, dynamic behaviour, and static functionality of the system and show how the design and refinement of constraints on these aspects can be separated, and how they can consistently form a whole model of the system. For verification and validation, we look at consistency between interactions and dynamic behaviour, component interaction protocols, static analysis and testing of functionality. We discuss how the activities of model construction, transformations, model verification and validation can be embedded into an existing commercial software development tool, MasterCraft [31]. We have selected this tool, 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. 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.

Overview The following Section 2 gives an overview on the main ideas and theme of our research on the rCOS methodology, and provides the formulation of the main concepts of model-drien development. In Section 3, we demonstrate, with our recent experience in the work on CoCOME case study, how the formalization of the concepts, models and techniques developed in rCOS can be integrated in a model-driven development process. The integration unifies the different formal techniques of verification and validation with correctness by design. We then discuss in Section 4 how we can enhance the industrial model-driven tool, MasterCraft, for the support of the integration of formal design, verification and validation into a practical engineering development process. Finally Section 5 summarizes our experience and discuss the plan for our future work.

2 The Basic Ideas Behind rCOS

The motivation of the research on rCOS is to provide a semantic foundation for *model driven development* in the combined framework of object-oriented and component-based framework. Practical software engineering shows that this is a promising approach to heighten productivity in software development while maintaining a high quality. It lets developers design systems at a higher level of abstraction using models or specifications of components which will be produced and integrated at a later implementation, assembly and deployment stage.

2.1 rCOS formulation of software concepts

A project using model driven development starts with a set of component specifications which may be given for previously developed components or be newly introduced for components that are to be developed later. The designers then proceed to

– build new components by applying component operators (connectors) to the given ones,
– build new components by programming glue processes,
– define application work-flows as processes that use services from components; and
– verification and validation are performed on components before composition and after composition

To provide formal support to such a development process, we formulate in rCOS the key notions as mathematic structures and study the rules for manipulation of these mathematic entities. These notions include *interfaces*, *contracts of interfaces*, *components*, *processes*, *compositions* and *refinement relations* on contracts, components and processes. In the next subsection, we give brief introduction to formulations.

Interfaces and contracts An interface I provides the syntactic type information of an interaction point of a component. It consists of two parts, the *data declaration section* denoted by $I.FDec$, that declares a set of variables with their types, and the *method declaration section*, denoted by $I.MDec$, that defines a set of method signatures each with the form $m(T_1\ in; T_2\ out)$. Interfaces are used for syntactic type checking. The current practical component technologies only provide syntactical aspects of interfaces and leaving the semantics of interfaces to informal naming schemes. This is obviously not enough for rigorous verification and validation. For example, a component with only syntactic interfaces shown in Fig. 1 has no information about its functionality or behavior.

A *contract* is a specification of the semantic details for the interface. However, different usages of the component in different applications under different environments may contain different details, and have different properties:

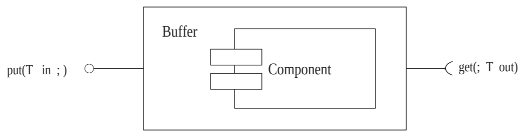

Fig. 1: A component with syntactic interface only

- An interface for a component in a sequential system is obviously different from one in a communicating concurrent system. A contract for the former only needs to specify the functionality of the methods, for example in terms of their pre- and post-conditions. A contract for the later should includes a description of the communicating protocol, for example in terms of interaction traces. The protocol specifies the order in which the interaction events happen.
- An interface for a component in a real-time application will need to provide the real-time constraints of services, but an untimed application does not.
- Components in distributed, mobile or internet-based systems require their interfaces to include information about their locations or addresses.
- An interface (component) should be stateless when the component is required to be used dynamically and independently from other components.
- A service component has different features from a middleware component.

Therefore, it is the contract of the interface that determines the external behavior and features of the component and allows the component to be used as a black box.

Based on the above discussion, rCOS defines the notion of an a contract of interface for a component as a description of what is needed for the component be *used* in building and maintaining software systems. The description of an interface must contain information about all the viewpoints among, for example functionality, behavior, protocols, safety, reliability, real-time, power, bandwidth, memory consumption and communication mechanisms, that are needed for composing the component in the given architecture for the application of the system. However, this description can be incremental in the sense that newly required properties or view points can be added when needed according to the application. Also, the consistency of these viewpoints should be formalizable and checkable. For this, rCOS is built on the Hoare and He's Unifying Theorems of Programming [16].

The minimal use of UTP In UTP, a *sequential program* (but possible non-deterministic) is represented by a *design* $D = (\alpha, P)$, where

- α denotes the set of state variables (called observables) of the program
- P is a predicate $p(x) \vdash R(x, x') \overset{def}{=} (ok \wedge p(x)) \Rightarrow (ok' \wedge R(x, x'))$, meaning that if the program is activated *ok* in a state where the *precondition* $p(x)$ holds

the execution will terminate *ok* in a state where the postcondition holds that post-state x' and the initial state x are related by relation R.

It is proven in UTP that the set of designs is closed under the classical programming constructs of *sequential composition*, *conditional choice*, *nondeterministic choice*, and fixed point of iterations. Refinement between design is defined as logical implications, and all the above operation on designs are monotonic with regard to refinements (i.e. the order of implication). These fundamental mathematic properties ensures that the domain of designs is a proper semantic domain for sequential programming languages. There is a nice link from the theory of designs to the theory of predicate transformers with the following definition:

$$\mathbf{wp}(p \vdash R, q) \overset{def}{=} p \wedge \neg(R; \neg q)$$

that define the *weakest precondition* of a design for a post condition q.

Concurrent and reactive programs, such as those specified by Back's action systems [5] or Lamport's Temporal Logic of Actions (TLA) [19], can be defined by the notion of *guarded designs*, written as $g\& D$ and defined by

$$(\alpha, \textbf{if } g \textbf{ then } P \textbf{ else } (true \vdash wait' \wedge v' = v))$$

The domain of guarded designs enjoys the same closure properties as that the domain. And refinement is defined as logical implication too.

The basic UTP has no notions of objects, classes, inheritance, polymorphism, and dynamic binding. For a combination of OO and component-based modelling, we have extend UTP to object-oriented programming [14].

Contracts of interfaces In the current version of rCOS, we only consider components in applications of concurrent and distributed systems, and a *contract* $Ctr = (I, Init, MSpec, Prot)$ specifies

- the allowable initial states by the initial condition $Init$,
- the synchronization condition g on each declared method and the functionality of the method by the specification function $MSpec$ that assigns each method to a guarded design $g\&D$.
- $Prot$ is called the *protocol* and is a set of sequences of call events; each is of the form $?op_1(x_1), \ldots, ?op_k(x_k)$. Notice a protocol can be specified by a temporal logic or a trace logic.

For example, the component interface in Fig. 1 does not say the buffer is a one place buffer. A specification of a one-place buffer can be given by a contract B for which

- The interface: $B_1.I = \langle q : Seq(int), put(item : int;), get(; res)\rangle$
- The initial condition: $B_1.Init = q = <>$

– The specification:

$$B_1.MSpec(put) = q =<> \&true \vdash q' =< item >$$
$$B_1.MSpec(get) = q \neq<> \&true \vdash res' = head(q) \wedge q' =<>$$

– The protocol: $B_1.Prot$ is a set of traces that is a subset of

$$\{e_1, \ldots, e_k \mid e_i \text{ is } ?put() \text{ if } i \text{ is odd and } ?get() \text{ otherwise}\}$$

.

The formulation of contracts supports separation of views, but the different views have to be consistent. A contract *Ctr* is *consistent*, if it will never enter a deadlock state if its environment interacts with it according to its protocol, that is, for all $\langle ?op_1(x_1), \ldots, ?op_k(x_k)\rangle \in Ctr.Prot$,

$$\mathbf{wp}\begin{pmatrix} Init;g_1\&D_1[x_1/in_1];\ldots;g_k\&D_k[x_k/in_k], \\ \neg wait \wedge \exists op \in MDec\bullet g(op) \end{pmatrix} = true$$

Note that this formalization takes both synchronization conditions and functionalities into account, as an execution of a method with its precondition falsified will diverge and a divergent state can cause deadlock too.

We have proven the following *theorem of separation of concerns*

Theorem 1. *(Separation of Concerns)*

1. *If* $Cons(I, Init, MSpec, Prot_i)$, *then* $Cons(I, Init, MSpec, Prot_1 \cup Prot_2)$
2. *If* $Cons(I, Init, MSpec, Prot_1)$ *and* $Prot_2 \subseteq Prot_1$, *then* $Cons(I, Init, MSpec, Prot_2)$
3. *If* $Cons(I, Init, MSpec, Prot)$ *and* $MSpec \sqsubseteq MSpec_1$, *then* $Cons(I, Init, MSpec_1, Prot)$

This allows us to refine the specification and the protocol separately.

We are now current working on an extension to the model of contracts for specification of the timing information of a component. An interesting and important point that we would like to make is that the notation for timing aspect at the contract level should be different from that used for the model of the design of components. At the contract level, we propose the use of interval based notation to describe the minimal time and maximal time $[t_e, T_e]$ that the environment has to wait when calling an interface method (that is the worst case execution time of the interface methods), and the minimal time and maximal time $[t_w, T_w]$ that the component is willing to wait to a method to be invoked. Zhou Chaochen's Duration Calculus [32] is an obvious choice for reasoning this interval based timing properties. However, for the design and for verification of the implementation of a component, clocks or timers in the timed automata model are more feasible. This indicates the use of different notations at different level of abstraction in a system development. A challenge is to link the clock time model for the design of components to the interval-based time model of its contract. Initial results on this work can be found [26].

Contract refinement A contract Ctr has a denotational semantics in terms of its *failure set* $\mathcal{F}(Ctr)$ and divergence set $\mathcal{F}(Ctr_1)$, that is same as the failure-divergence semantics for CSP (but we do not use the CSP language) [7]. Ctr_1 is *refined* by contract Ctr_2, denoted by $Ctr_1 \sqsubseteq Ctr_2$, if the later offers the same provided methods, $Ctr_1.MDec = Ctr_2.MDec$, is not more likely to diverge than the former, $\mathcal{D}(Ctr_1) \supseteq \mathcal{D}(Ctr_2)$, and not more likely to deadlock than the former, $\mathcal{F}(Ctr_1) \supseteq \mathcal{F}(Ctr_2)$. We have established a complete proof techniques of refinement by simulation.

Theorem 2. *(Refinement by Simulation) $Ctr_1 \sqsubseteq Ctr_2$ if exists a total mapping $\rho(u, v') : FDec_1 \longrightarrow FDec_2$ such that*

1. *$Init_2 \Rightarrow (Init_1; \rho)$*
2. *$\rho \Rightarrow (guard_1(op) = guard_2(op))$ for all $op \in MDec_1$.*
3. *for each $op \in MDec_1$, $MSpec_1(op); \rho \sqsubseteq \rho; MSpec_2(op)$*

Similarly, contract refinement can also be proved by a surjective upward simulation [7].

Theorem 3. *(Completeness of Simulations)If $Ctr_1 \sqsubseteq Ctr_2$, there exists a Ctr such that*

$$Ctr_1 \preceq_{up} Ctr \preceq_{down} Ctr_2 \qquad Ctr_1 \sqsubseteq Ctr \sqsubseteq Ctr_2$$

$\preceq_{up}$ and $\preceq_{down}$ denote upwards and downwards simulations, respectively.

Components A component is an *implementation* of a contract. Formally speaking, a *component* is turple $C = (I, Init, MCode, PriMDec, PriMCode, InMDec)$, where

– $MCode$ and $PriMCode$ map a public method and a private method m to a guarded command $g_m \rightarrow c_m$,
– $InMDec$ is the set of required methods in the code, called the *required interface.*

The *semantics* $[\![C]\!]$ is a function that calculates a contract for the provided interface for any given contract $InCtr$ of the required interface

$$[\![C]\!](InCtr) \overset{def}{=} ((I, MSpec), Init, PriMDec, PriMSpec)$$

where the specification is calculated from the semantics of the code, following the calculus established in UTP.

A component C_1 is refined by another component C_2, denoted by $C_1 \sqsubseteq C_2$ if

1. the later provides the same services as the former, $C_1.MDec = C_2.MDec$
2. the later requires the same services as the former $C_1.InMDec = C_2.InMDec$, and
3. for any given contract of the required interface, the resulting provided contract of the later is a refinement of that of the former, $C_1(InCtr) \sqsubseteq C_2(InCtr)$, holds for all input contracts $InCtr$.

Note that the notion of component refinement is used for both *component correctness by design* and component *substitutability* in maintenance. One of the major objectives of rCOS is to prove *design patterns* as refinement rules, and automate refinement rules as *model transformations*. We hope this will help to reduce the amount of verification required.

Simple connectors To support the development activity, the semantic framework also needs to define operators for connecting components, resulting in new contracts, constructs for defining glue processes, and constructs for defining processes. In summary, the framework should be *compositional and support both functional and behavioral specification.* In rCOS, simple connectors between components are defined as component compositions. These include *plugging* (or *union*), *service hiding*, *service renaming*, and *feadback*. These compositions are shown in Figs. 2-4.

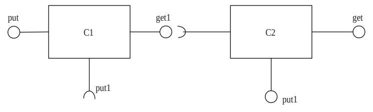

Fig. 2: Plug Composition

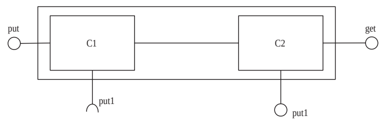

Fig. 3: Hiding after Chaining

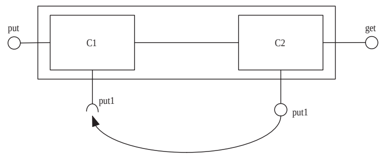

Fig. 4: Feedback

2.2 Coordination

From an external point of view, components provide a number of methods; but does not themselves activate the functionality specified in the contracts; we need active entities that implements a desired functionality by coordinating the sequences of method calls. In general, these active entities do not share the three features of components [13].

In [7], we introduce *processes* into rCOS. Like a component, a process has an interface declaring its own local state variables and methods, and its behavior is specified by a process contract. 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 to required services or to wait for a call to its provided services. For such an active process, we cannot have separate contracts for its provided interface and required interface, because we cannot have separate specifications of outgoing calls and incoming calls [13]. So a process only has an interface and its associated contract (or code). For simplicity, but without losing expressiveness, we assume a process like a Java thread does not provide services and only calls methods provided by components. Therefore, processes can only communicate via shared components. Of course, a component can also communicate with another component via processes, but without knowing component that it is communicating with.

Let C be the parallel composition of a number of disjoint components C_i, $i = 1 \dots k$. A glue program for C is a process P that makes calls to a set of X of provided methods of C. The composition $C\|\ [X]P$ of C and P is defined similarly to the alphabetized parallel composition in CSP [30] with interleaving of events. The gluing composition is defined by hiding the synchronized methods between the component C and the process P. We have proven that $(C\|\ [X]P)\backslash X$ is a component, and studied the algebraic laws of the composition of processes and components. The glue composition is illustrated in Fig. 5, where in Fig. 5(a) $C1$ and $C2$ are two one-place buffers and P is a process that keeps getting the item from $C1$ and putting it to $C2$. In Fig. 5(b), the *get* of $C1$ and *put* of $C2$ are synchronized into an atomic step by component M; and M proves method $move()\{get1(;y); put2(y;)\}$, that process P calls.

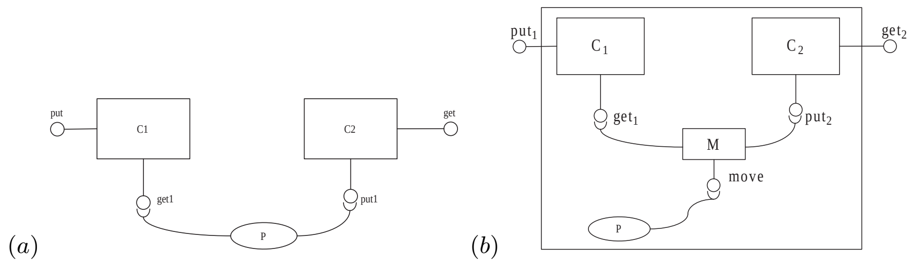

Fig. 5: (a) Gluing two one-place buffers forms a three-place Buffer, (b) Gluing two One-place buffers forms a two-place buffer

An *application program* is a set of parallel processes that make use of the services provided by components. As processes only interact with components via the provided interfaces of the components, interoperability is thus supported by the contracts which define the semantics of the common interface description language (IDL), even though components, glue programs and application programs are not implemented in the same language. Analysis and verification of an application program can be performed in the classical formal frameworks, but at the level of contracts of components instead of implementations of components. The analysis and verification can reuse any proved properties about the components, such as divergence freedom and deadlock freedom without the need to reprove them.

2.3 Object-orientation in rCOS

The variables in the field declaration section can be of object types. This allows us to apply OO techniques the design and implementation of a component. In our earlier work [14], we have extended UTP to formal treatment of OO program and OO refinement. This is summarized as follows.

Classes In rCOS, we write a class specification in the following format:

$$
\begin{array}{ll}
\textbf{class} & C\ [\textbf{extends}\ D]\{ \\
\textbf{attr} & T_1\ x = d, \ldots, T_k\ x = d \\
\textbf{meth} & m(T\ \mathit{in};\ V\ \mathit{return})\ \{ \\
 & \quad \textbf{pre:} \quad c \vee \ldots \vee c \\
 & \quad \textbf{post:} \quad (R; \ldots; R) \vee \ldots \vee (R; \ldots; R) \\
 & \qquad\qquad \wedge \ldots\ldots \\
 & \qquad\qquad \wedge\ (R; \ldots; R) \vee \ldots \vee (R; \ldots; R)\ \} \\
 & \ldots\ldots \\
\textbf{meth} & m(T\ \mathit{in};\ V\ \mathit{return})\ \{\ldots\ldots\ \} \\
 & \ldots\ldots \\
\textbf{invariant} & \mathit{Inv}\ \}
\end{array}
$$

The initial value of an attribute is optional, and an attribute is assumed to be public unless it is tagged with reserved words *private* and *protected*. If no initial value is declared it will default to *null*.

Each c in the precondition represents a condition to be checked; it is a conjunction of primitive predicates.

A *design* $p \vdash R$ for a method is written as **Pre** p and **Post** R. And R in the post-condition is of the form $c \wedge (le' = e)$, where c is a condition, le an *assignable expression* and e an expression. An assignable le is either a primitive variable x, or an attribute name a or $le.a$. An expression e can be a logically specified expression such as the greatest common divisor of two given integers.

We allow the use of *indexed conjunction* $\forall i \in I : R(i)$ and *indexed disjunctions* $\exists i \in I : R(i)$ for a finite set I. These would be the quantifications if the index set is infinite. The reader can see the influence of TLA$^+$ [19], UNITY [6] and Java on the above format.

OO refinement OO design is to design object interactions so that objects interacts with each other to realize the functionality specified in the class declarations. In rCOS, we provide three levels of refinement:

1. Refinement of a whole object program. This may involve the change of anything as long as the behavior of the main method with respect to the global variables is preserved. It is an extension to the notion of data refinement in imperative programming, with a semantic model dealing with object references, method invocation, and polymorphism. In such a refinement, all non-public attributes of the objects are treated as local (internal) variables.
2. Refinement of the class declaration section: $\mathit{Classes}_1$ is a refinement of *Classes* if $\mathit{Classes}_1 \bullet \mathit{main}$ refines $\mathit{Classes} \bullet \mathit{main}$ for all *main*. This means that $\mathit{Classes}_1$ supports at least as many services as *Classes*.
3. Refinement of a method of a class in *Classes*. Obviously, $\mathit{Classes}_1$ refines *Classes* if the public class names in *Classes* are all in $\mathit{Classes}_1$ and for each public method of each public class in *Classes* there is a refined method in the corresponding class of $\mathit{Classes}_1$.

Very interesting results on completeness of the refinement calculus are available in [23].

In an OO design there are mainly three kinds of refinement: *Delegation of functionality* or *responsibilities*, *attribute encapsulation*, and *class decomposition*.

Delegation of functionality Assume that C and C_1 are classes in *Classes*, $C_1\ o$ is an attribute of C and $T\ x$ is an attribute of C_1. Let $m()\{c(o.x', o.x)\}$ be a method of C that directly accesses and/or modifies attribute x of C_1. Then, if all other variables in the method c are accessible in C_1, we have $Classes \sqsubseteq Classes_1$, where $Classes_1$ is obtained from *Classes* by changing $m()\{c(o.x', o.x)\}$ to $m()\{o.n()\}$ in class C and adding a fresh method $n()\{c[x'/o.x', x/o.x]\}$. This is also called the *expert pattern of responsibility assignment.*

This rule and other refinement rules can prove big-step refinement rules, such as the following **expert pattern**, that will be repeatedly used in the design of POS.

Theorem 4 (Expert Pattern). *Given a list of class declarations Classes and its navigation paths* $r_1.\dots.r_f.x$ *(denoted by le),* $\{a_{11}.\dots.a_{1k_1}.x_1, \dots, a_{\ell 1}.\dots.a_{\ell k_\ell}.x_\ell\}$*, and* $\{b_{11}.\dots.b_{1j_1}.y_1, \dots, b_{t1}.\dots.a_{tj_t}.y_t\}$ *starting from class C, let* $m()$ *be a method of C specified as*

$$\begin{aligned} C :: m()\{\quad & c(a_{11}.\dots.a_{1k_1}.x_1, \dots, a_{\ell 1}.\dots.a_{\ell k_\ell}.x_\ell) \\ & \wedge\ le' = e(b_{11}.\dots.b_{1s_1}.y_1, \dots, b_{ts1}.\dots.b_{ts_t}.y_t)\ \} \end{aligned}$$

Then Classes can be refined by redefining $m()$ *in C and defining the following fresh methods in the corresponding classes:*

$$\begin{array}{ll} C :: & check()\{return'{=}c(a_{11}.get_{\pi_{a_{11}}x_1}(), \dots, a_{\ell 1}.get_{\pi_{a_{\ell 1}}x_\ell}())\} \\ & m()\{\mathbf{if}\ check()\ \mathbf{then}\ r_1.do\text{-}m_{\pi_{r_1}}(b_{11}.get_{\pi_{b_{11}}y_1}(), \dots, b_{s1}.get_{\pi_{b_{s1}}y_s}())\} \\ T(a_{ij}) :: & get_{\pi_{a_{ij}}x_i}()\{return'{=}a_{ij+1}.get_{\pi_{a_{ij+1}}x_i}()\}\ (i : 1..\ell, j : 1..k_i - 1) \\ T(a_{ik_i}) :: & get_{\pi_{a_{ik_i}}x_i}()\{return'{=}x_i\}\ (i : 1..\ell) \\ T(r_i) :: & do\text{-}m_{\pi_{r_i}}(d_{11}, \dots, d_{s1})\{r_{i+1}.do\text{-}m_{\pi_{r_{i+1}}}(d_{11}, \dots, d_{s1})\}\ \ for\ i : 1..f-1 \\ T(r_f) :: & do\text{-}m_{\pi_{r_f}}(d_{11}, \dots, d_{s1})\{x' = e(d_{11}, \dots, d_{s1})\} \\ T(b_{ij}) :: & get_{\pi_{b_{ij}}y_i}()\{return'{=}b_{ij+1}.get_{\pi_{b_{ij+1}}y_i}()\}\ (i : 1..t, j : 1..s_i - 1) \\ T(b_{is_i}) :: & get_{\pi_{b_{is_i}}y_i}()\{return'{=}y_i\}\ (i : 1..t) \end{array}$$

where $T(a)$ *is the type name of attribute* a *and* π_{v_i} *denotes the remainder of the corresponding navigation path* v *starting at position* j*.*

This pattern informally represents the fact that a computation is realized by obtaining the data that distributed in different objects via association links and then delegating the computation tasks to the target object whose state is required to change.

If the paths $\{a_{11}.\dots.a_{1k_1}.x_1, \dots, a_{\ell 1}.\dots.a_{\ell k_\ell}.x_\ell\}$ have a common prefix, say up to a_{1j}, then class C can directly delegate the responsibility of getting the x-attributes and checking the condition to $T(a_{ij})$ via the path $a_{11}.\dots, a_{ij}$ and then follow the above rule from $T(a_{ij})$. The same rule can be applied to the b-navigation paths.

The expert pattern is the most often used refinement rule in OO design. One feature of this rule is that it does not introduce more couplings by associations between classes into the class structure. It also ensures that functional responsibilities are allocated to the appropriate objects that *knows* the data needed for the responsibilities assigned to them.

Encapsulation The *encapsulation rule* says that if an attribute of a class C is only referred directly in the specification (or code) of methods in C, this attribute can be made a *private attribute*; and it can be made *protected* if it is only directly referred in specifications of methods of C and its subclasses.

Class decomposition During an OO design, we often need to decompose a class into a number of classes. For example, consider classes $C_1 :: D\ a_1$, $C_2 :: D\ a_2$, and $D :: T_1\ x, T_2\ y$. If methods of C_1 only call a method $D :: m()\{...\}$ that only involves x, and methods of C_2 only call a method $D :: n()\{...\}$ that only involves y, we can decompose D into two $D_1 :: T_1\ x; m()\{...\}$ and $D_2 :: T_2\ y; n()\{...\}$, and change the type of a_1 in C_1 to D_1 and the type of a_2 in C_2 to D_2. There are other rules for class decomposition in [14].

An important point here is that the expert pattern and the rule of encapsulation can be implemented by automated model transformations. In general, transformations for structure refinement can be aided by transformations in which changes are made on the structure model, such as the class diagram, with a diagram editing tool and then automatic transformation can be derived for the change in the specification of the functionality and object interactions. For details, please see our work in [23].

3 Integrating rCOS Support into Model-Driven Development Process

In a realistic project there are more *activities* than just design. These activities are performed by project team members in different *roles*, such as *Administrator, Analysis Modeler, Architecture Modeler, Design Modeler, Construction Manager, Construction Programmer, Model Manager*, and *Version Manager* [31]. The concepts of activities and roles define at which point various models, that are also informally called *artifacts*, are produced by which roles, and what different analysis, manipulation, checking and verification are performed, with different tools. The concept of roles is also useful for the control of the work flow in that different roles are allowed to access and modify certain models in the development environment. These concepts and ideas have been implemented in the industrial tool, MasterCraft, for model transformation [31]. In this section, we use the our experience with the in the recent work on the Common Component Modelling Example (CoCoME) to show how the rCOS methodology can be integrated into a model-driven development processes in supporting the development activities. We first introduce the modelling example, that is followed by a summary of the application of rCOS.

3.1 POST—the common modelling example

The point of sale system (POST) was originally used as a running example in Larman's book [20] to demonstrate the concepts, modeling and design of object-oriented systems. An extended version is now being used as the case study in the Common Component Modeling Contest (CoCOMe) [4].

POST is a computerized system typically used in a retail store. It records sales, handles both cash payments and credit card payments as well as inventory update. Furthermore, the system deals with ordering goods and generates various reports for management purposes. The system can be a small system, containing only one terminal for checking out customers and one terminal for management, or a large system that has a number of terminals for checking out in parallel, or even a network of these large systems to support an enterprise of a chain of retail stores. The whole system includes hardware components such as computers and bar code scanners, card readers, and a software to run the system. To handle credit card payments, orders and delivery of products, we assume a *Bank* and a *Supplier* that system POST with which can interact.

In the common modelling excise requires each team to work on a common informal description of the system, and carry out a component-based modelling and design. Various of aspects should be modelled and analysed, including functionalities, interactions, middlewares, and extra-functionalities (also known as non-functional requirements) such as timing. It is required to generate code for the implementation too.

The problem description that we received is largely based on *use case descriptions*. There can be many use cases for this system, depending on what business processes the client of the system want the system to support. One of the main use cases is *processing sales*, that is denoted by the use case **UC1**: *Process sales*. An informal description can be given as follows.

This use case can perform either *express checkout process* for customers with only a few items to purchase, or a *normal checkout process*. The main course of interactions between the actors and the system is described as follows.

1. The *cashier* sets the checkout mode to for express check out or for normal check out. The system then sets the *displaylight* to *green* or *yellow* accordingly.
2. This use case starts when a *customer* comes to the *checkout point* with their *items* to purchase.
3. The cashier indicates the system to handle a new sale.
4. The cashier enters all the items, either by typing or scanning in the *bar code*, if there is more than one of the same item, the cashier can also enter the *quantity*. The system records each item and its quantity and calculates the subtotal.
   In the express checkout mode, only a limited number of items are allowed to checkout.
5. At the end of inputting the items, the *total* of the sale is calculated. The cashier tells the customer the total and asks her to pay.
6. The customer can pay by cash or a credit card:
   (a) If by cash, the cashier enters the amount received from the customers, and the system record the *cash payment* amount and calculate the change. The cashier gives the change to the customer.
   (b) If the customer chooses to pay by a credit card, the cashier enters the card information (manually or by the card reader). The system sends the credit payment to the *bank* for *authorization*. The payment can only be made if a positive authorization reply is received.

   The inventor of the sold items are updated and the completed sale is logged in the *store*.
7. The customer leaves with the items they purchased at the end of the process.

There are exceptional courses of interactions. For example, the entered bar code is not known in the system, the customer does not have enough money for a cash payment, or the authorization reply is negative. Systems needs to provide means of handling these exception cases, such as canceling the sale or change to another way of paying for the sale. At the requirements level, we capture these exceptional conditions as preconditions.

Other use cases include **UC2**: *Order products*, that orders products from the supplier; **UC3**: *Manage inventory*, that include changing the amount of an item (after receiving deliveries from the product supplier), changing the price of a product, and adding a new product, and deleting a new product; **UC4**: *Produce monthly reports on sales* that is to show the reports of all sales in the last 30 days and information of profit and loss; and **UC5**: *Produce stock reports* , that produces the reports on stock items.

3.2 Development of POST with rCOS

There has been a wide view that object-oriented and component-based design should be bottom up. We in fact take a use-case driven, incremental and iterative Rational Unified Development Process [18].

The sketch of the development process In each iteration, a number of use cases are captured, modeled and analysed at the requirements stage. Each use case is modeled as a contract of a component that provides services to the actors of the use case. The fields of the contract declare the domain objects involved in the realization of the use case. The classes of these objects are organized as a class diagram representating the structural view of the data and objects of the components. The contracts should be analyzed and the consistency of the contracts should be checked.

The contracts of the use case components are then designed and organised into bigger components to form the component-based architecture for the application software components with identified *object-oriented interfaces.* We call this step the *logical design* of the iteration. This involves object-oriented refinement of each use case component, identifies interactions among objects in different components, and compose components (i.e. use cases) by simple connectors. The resulting model is the *logical component model.*

The model of logical design should be further refined by class decomposition, data encapsulation and refactoring. We call this step the *detailed design.* The detailed design also involves replacing the object-oriented interfaces with concrete and appropriate interaction mechanisms (or middlewares) such as RMI, CORBA or shared event channels.Verification and validation, such as runtime checking or testing (unit testing), can be applied to components before and after introducing the concrete middlewares.

Code can be constructed for each component and static analysis, unit testing and runtime checking can be done on the components.

Before or after coding, the design of the GUI, the software controller of the hardware devices and their interactions with the application software components can be modeled and designed. This is done in a purely event-based model following the theory of embedded system design. The business components, GUI components hardware controllers and middlewares are integrated and deployed.

Requirements modelling of POST A use case is modelled as a contract of a component, that corresponds the concepts of *use case controller class* in our earlier object-oriented modelling [8]. To help practical software engineers to understand the formal models, the protocol of a use case contract is illustrated by a UML sequence diagram that defines all the possible traces of the interaction between the actors and the system in the use case. The guarded design specification of each interface method is further divided into the guard, the control state transition, and the data functionality. The guards and the control state transitions are shown by a UML state diagram, and the data functionality of a method is specified as unguarded design. For such a style of modelling and their formal integration, we refer to our paper [8]. The protocol of use case process a sale **UC1** is modelled by the sequence diagram in Fig. 6 and its state diagram is given in Fig. 7.

The specification of the functions of the use case and and the component invariant are given as follows.

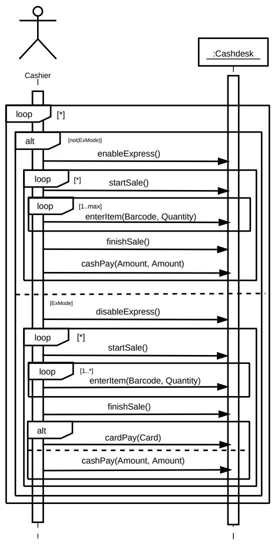

Fig. 6: Sequence Diagram

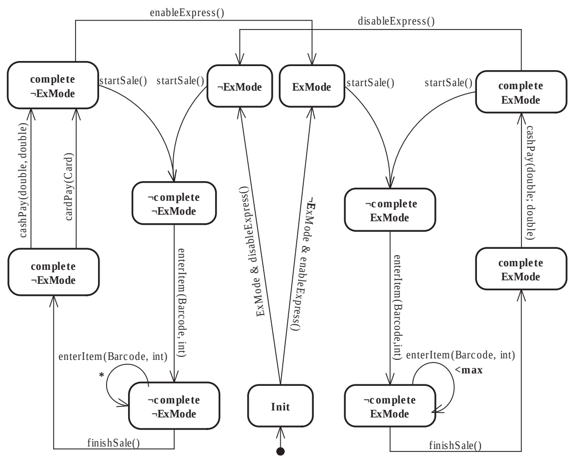

Fig. 7: Design Sequence Diagram

Use Case UC1: Process Sales

Component *Cashdesk*{

meth $enableExpress()$ {
Pre: $true$;
Post: $light.display' = green$ };

meth $disableExpress()$ {
Pre: $true$;
Post: $light.display' = yellow$ }

meth $startSale()$ {
Pre: $true$;
Post: /** a new *sale* is created and its *lines* initialized to empty, and its dates correctly recorded **/;
$sale' = Sale.New()$}

meth $enterItem(long\ c, double\ q)$ {
Pre: $store.cat.find(c) \neq null$ /**the input barcode c is valid **/
Post: /** a new *line* is created with its *barcode* and *quantity* set to c and q, **and****/;
$\wedge\ line' = LineItem.New(c/barcode, q/quantity)$
/**the *subtotal* of the *line* is set, **and** **/
$\wedge\ line.subtotal' = store.cat.find(c).price \times q)$
/**add *line* to the current *sale* **/
$\wedge\ sale.lines.add(line')$ }

meth $cashPay(double\ a; double\ c)$ {
Pre: $a \geq sale.total$ /** amount is no less then the total **/
Post: /** the *Cashpayment* of the *sale* is created, **and****/
$\wedge\ sale.pay' = CashPayment.New($
$a/amount, a - sale.total/change)$
/** the change is returned, **and then** the completed *sale* is logged in *store*, **and** **/
$\wedge\ c' = a - sale.total; store.sales.add(sale)$
/** the inventory is updated **/
$\wedge\ \forall l \in sale, lines, \forall p \in store.cat : ($
if $p.barcode = l.barcode$ **then** $p.amount' = p.amount - l.quantity)$ }

meth *finishSale*() {
    Pre: *true*;
    Post: /** *sale* is set to *complete*, **and**;
    $\wedge$ $sale.complete' = true$
    /** *sale*'s *total* is calculated **/
    $\wedge$ $sale.total' = addAll[[l.subtotal | l \in sale.lines]]$ }
meth *cardPay*(*Card c*) {
    Pre: /** the card is valid **/
    $valid(c, sale.total)$ /**authorized by the bank **/;
    Post: /** the *CardPayment* of the *sale* is created, **and then** the completed *sale* is logged in *store*, **and** **/
    $\wedge$ $sale.pay' = CardPayment.New(c/card)$**;**
      $store.sales.add(sale)$
    /** the inventory is updated **/
    $\wedge$ $\forall l \in sale.lines, \forall p \in store.cat :$ (
      if $p.barcode = l.barcode$ **then**
      $p.amount' = p.amount - l.quantity$) }
}

The structure of the data and classes of the objects are declared as class declarations in rCOS and can be illustrated by a UML class diagram. Then the state space of the component is the set of the object diagrams of the class diagram. Fig. 8 shows the class diagram of use case **UC1** and Fig. 9 is an example of an object diagram.

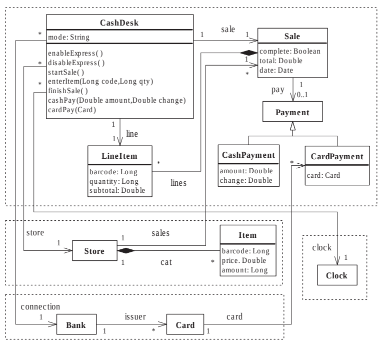

Fig. 8: Class Diagram of Process Sale

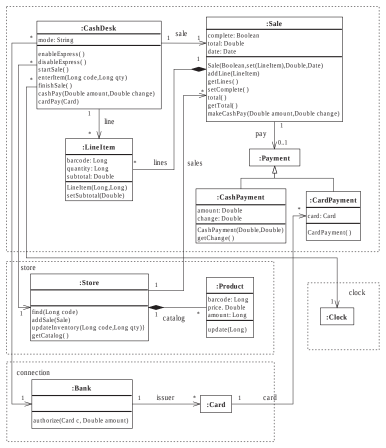

Fig. 9: Object Diagram of Process Sale

The execution of an invocation to an interface method changes from one object diagram to another [14, 23]. The behaviour of the use case components (the methods used above) will be implemented in an *abstract class*, and the used methods and arguments indicate its abstract interface:

```
public abstract class Cashdesk implements SalesHandler Interface{
  protected boolean exmode;
  public abstract void enableExpress();
  public abstract void disableExpress()
  public abstract void startSale();
  public abstract void enterItem(Barcode code, int quantity);
  public abstract void finishSale();
  public abstract void cardPay(Card c, double a);
  public abstract void cashPay(double a);
}
```

Requirements consistency Static consistency between methods in the diagrams and the functional specification, their types, and navigation paths must be consistent. This step is usually done by some tools like a compiler, but is done manually in the case study due to a lack of machine readable specifications.

Dynamic consistency ensures that the separately specified behavior in the sequence diagram, the state diagram and the trace are consistent. Informally, the consistency must ensure that whenever the actors follow the interaction protocol defined by the sequence diagram, the interactions will not be blocked by the system, i.e. no deadlock should occur. Formally speaking, this requires that the traces are accepted by the state machine defined by the state diagram. Also, the sequence diagram should completely define the set of traces that can be accepted by the state diagram. While, the sequence diagrams specifies the traces in a denotational manner, the state diagram decries the flow of control in an operational semantics and thus model checking and simulation can be easily applied. The state diagram allows verification of both safety and liveness properties.

As all three specifications mechanisms are based on *regular* techniques and can be interpreted as defining languages of traces, we translate them manually into CSP specifications and use the FDR model checker to prove *trace equivalence* of the sequence and the state diagram. Likewise, we can generate PROMELA specifications for the SPIN model checker to check additional properties such as certain liveness or application specific properties.

Logical design The logical design step has two kinds of activities. First each use case contract is refined from its functional specification through application of design patterns, the Expert Pattern [9] in particular. This step delegates the functionality responsibilities to the internal domain objects (i.e. those of the fields). This derives a refinement of the use case sequence diagram into a *design sequence diagram*. For example, applying the expert pattern to the use case operation of **UC1** we can refine it to the design sequence diagram shown in Fig. 10. We can specify the other use cases and refine them in the same way. For the formal refinement of the use cases in rCOS, we refer the reader to the rCOS solution to CoCoME [**?**].

After the initial object-oriented refinement we can identify further components. For use case **UC1**, we single out the component $<< Clock >>$ and $<< Bank >>$ from the component $<< SalesHandler >>$. We also, compose the use cases for "order products", "manage inventory items", "produce sales reports" and "produce stock reports" into one component called $<< Inventory >>$. From the design sequence diagrams of the use cases (and formally the refined design of the use case operations specified in rCOS), we can organize the interaction among objects from the different components into provided and required interfaces of the identified components. This then transforms the model of the use case contracts into a logical component architecture as shown in

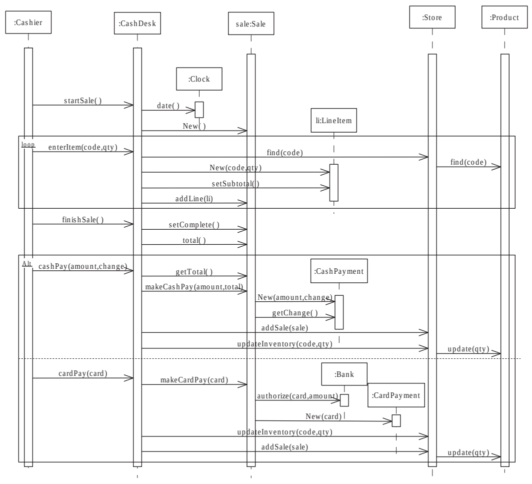

Fig. 10: Design Sequence Diagram of Process Sale

Fig. 11. The rCOS specification of the refined component $<< SalesHandler >>$ can be given as

component SalesHandler **required interface** ClockIf { date() }
required interface BankIf { authorize(..) } **required interface**
StoreIf { update(..), find (..), addSale(..) } **provided interface**
SaleIf { startSale , enterItem, finishSale , cashPay, cardPay }
protocol { ([?enableExpress (?startSale (?enterItem)$^{(max)}$?finishSale ?cashPay)*
| ?disableExpress (?startSale (?enterItem)* ?finishSale
[?cardPay | ?cashPay])*])* }
class *Cashdesk* **implements** SaleIf

This notation thus combines aspects of an rCOS *component* (provided/required interface and class implementing the provided methods) and *contract* (protocol). Call-ins in the protocol are indicated by a question mark. A *process* can be recognized by a protocol which starts with a call-out, denoted by an exclamation mark following the method name. Further decomposition of the component $<< Inventory >>$ into the three layer architecture consisting of << *Application* >>, data representation component << *Store* >> and << *Database* >> is shown in Fig. 12.

Notice that in the logical component models, interfaces are object-oriented interfaces, meaning that the interactions are through direct object method invocations.

Detailed design In the detailed design, refinement translates the specifications in the logical design into an object-based programming language resembling Java. In this step, class decomposition, refactoring [12] and data encapsulation, that proved as refinement rules in the object-oriented rCOS [14], can be applied.

Significant algorithms for specifications of methods of classes are designed. Such a method usually does not need to call methods outside its owning class. The specification of such a algorithms often uses quantifications over elements of an multi-object (or a container object). In rCOS, this is resolved through standard patterns like iteration,

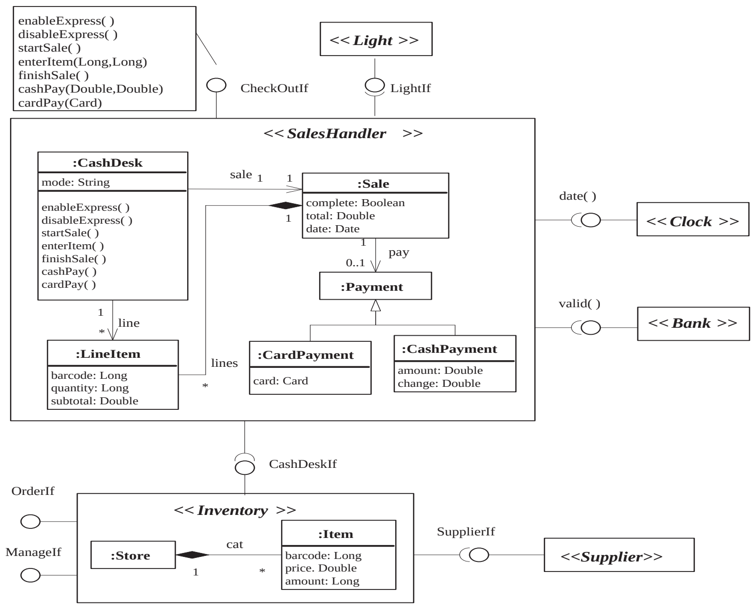

Fig. 11: The logical component-based model

although for example for database accesses it might be possible to refine them into SQL queries. The correctness of those patterns has been formally proved in previous rCOS literature.

This representation allows almost direct translation into Java. We invite the reader to observe the introduction of the intermediate classes which finally break down the *store.catalog.find()* in class *Cashdesk* down to the *set*-implementation, which is given again as a purely functional specification, under the assumption that a corresponding data structure is available in the target language:

```
class Cashdesk::      enterItem(Barcode code, int qty) {
                       if find(code) ≠ null then {
                         line:=LineItem.New(code, qty);
                         line.setSubtotal(find(code).price × qty);
                         sale.addLine(line) }
                      else { throw exception e }   }
                      find(Barcode code; Product r) {r:=store.find(code)}
class Store::         find(Barcode code; Product r) {r:=catalog.find(code)}
class set(Product):: find(Barcode code; Product returns)
                      Pre  ∃p : Product • (p.barcode = code ∧ contains(p))
                      Post returns.barcode' = code
class Sale::          addLine(LineItem l) {lines.add(l)}
class LineItem::      setSubtotal(double a) {subtotal:=a }
```

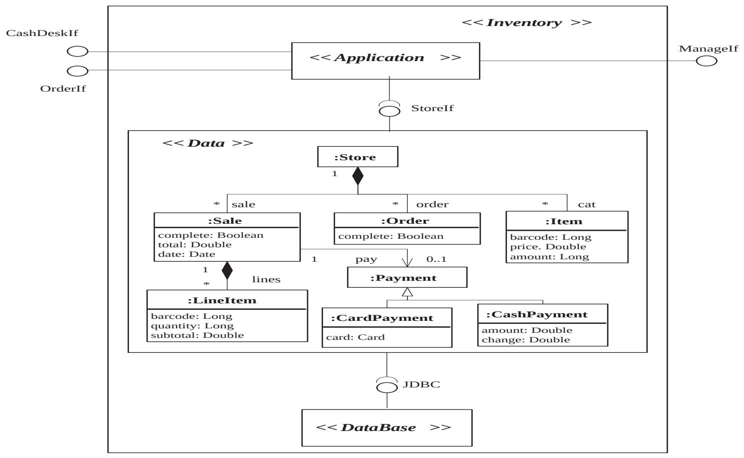

Fig. 12: The components of $<< Inventory >>$

```
class Cashdesk:: finishSale()    { sale.setComplete(); sale.setTotal() }
class Sale::     setComplete()   { complete:=true }
                 setTotal()      { total:=lines.sum() }
```

Separately, abstract interfaces for the classes containing a translation of the pre- and post-conditions into the Java Modelling Languages (JML) have been carried out. These can be checked at runtime, and we plan to use them for further static analysis in future work: The JML code snippet of the *enterItem()* design is shown on the left of Fig. 13. Notice that the code in the dotted rectangle gives the specification of the exception.

```
/*@ public normal_behaviour
  @   requires (\exists Object o; theStore.theProductList.contains(o);
  @            ((Product)o).theBarcode.equals(code)); ...
  @   ensures    theLine != \old(theLine) &&
  @     theLine.theBarcode.equals(code) &&...
  @ also
  @ public exceptional_behaviour
  @     requires !(\exists Object o; theStore.theProductList.contains(o);
  @              ((Product)o).theBarcode.equals(code));
  @     signals_only Exception;
  @*/
public void enterItem(Barcode code, int qty) throws Exception;
```

```
public void enterItem(Barcode code, int qty)
         throws Exception{
   if (find(code) != null) {
     line = new LineItem(code, qty);
     line.setSubtotal(find(code).price * qty);
     sale.addLine(line);
     t = true;
   } else {
     throw new Exception();
   }
}
```

Fig. 13: JML Specification and Implementation.

In the detailed design, some of the object-oriented interfaces are replaced by appropriate interaction mechanisms and middlewares, for example

– We keep the interface *StoreIf* between the application layer and the data representation layer as an oo interface.
– As all the *SalesHandler* instances share the same inventory, we can introduce a connector by which that the cash desks get product information or request the

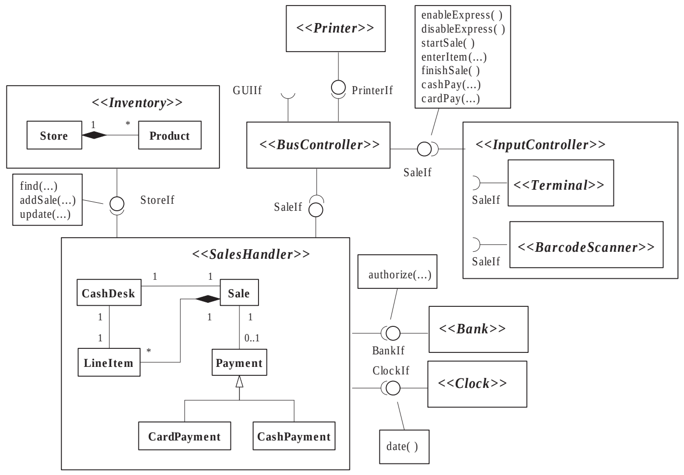

Fig. 14: Component diagram of Process Sale

inventory to update information of a product by passing a product code. This can be implemented asynchronously using an event channel.

– The interaction between the *SalesHandler* instances and *Bank* can be made via RMI or CORBA.
– The interaction between the *Inventory* instance and the *Supplier* can be made via RMI or CORBA.

Design of GUI and controllers of hardware devices In our approach, we keep the design of application independent from the design of the GUI, so that we do not need to change the application. The GUI design only concerned about how to link methods of GUI objects to interface methods of the application components to delegate the operation requested and to get the information that are needed to display on the GUI. So in general, the application components should not call methods of the GUI objects. Also, no references should be passed between application components and GUI components (the so called service-oriented interfaces should be used). This requires that all information that are to display on the GUI should be ready in the application components and corresponding interface operations should be provided by the application components to the GUI components. There are existing GUI builders can be used.

Each *SalesHandler* instance is connected to a bar code scanner, a card reader, a light, a cash box, and a printer. The hardware controllers also communicate with the GUI objects. For example, when the cashier presses the *startSale* button at his cash desk, the corresponding *SalesHandler* instance should react and the printer controller should also react to start to print the header of the receipt. The main communication can be done by using events which are sent through event channels. An obvious solution is that each *SalesHandler* has its own event channel, called *checkOutChannel*. This channel is used by the *CheckOut* instance to enable communication between all device

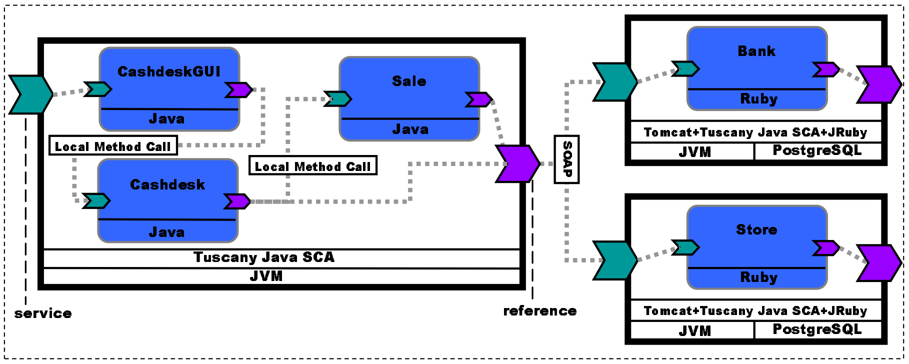

Fig. 15: SCA based Implementation

controllers, such as *LightDisplayController*, *CardReaderController* and the GUIs. The component, the device controllers and the GUI components have to register at their *checkOutChannel* and an event handlers have to be implemented and a message middleware, such as JMS, are needed to call the event event handlers. All the channels can be organized as component called *EventBus*. Component-based model of the system with the hardware components is shown in Fig. 14.

After all the components discussed in the previous subsections are designed and coded, the system is ready for deployment, that we leave out of this paper.

Service component architecture based implementation Based on the design of classes and components, additionally to the Java implementation of the business log, we implemented the system using Service Component Architecture (SCA) [3] and its supporting platform Tuscany Java SCA [1]. SCA provides a language-independent way to define and compose service components in the system, and it also supports different language-specific ways to implement the components. The SCA component specification can be generated from rCOS component description. The component implementation can be coded with respect to the component function features and the corresponding rCOS class design. We have implemented a prototype CoCoME system that contains six different distributed applications. The system components and their implementation and running information included in the sale process are shown in Fig. 15. The bold black rectangles represent the independent applications that can be deployed and run on different machines. The *Bank* and *Store* components are published as Web services, whose WSDL method description can be generated from the method definitions in rCOS component description, and the SOAP protocol binding on HTTP is used for the communications between applications. In addition, the *Bank* and *Store* components currently will create a new component instance for handling each client request.

During the development process, from the rCOS design, the most appropriate implementation technology can be used for different components, such as the *Ruby* language for the *Store* component, and we can also build the application based on the generated Java implementation from the rCOS design. The implementation only took two days. This process also corresponds to the spirits of Agile Software Development (Extreme Programming and Adaptive Software Development) [27].

4 Enhance Industrial Tool Support: MasterCraft

MasterCraft [31] is developed by TRDCC 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.

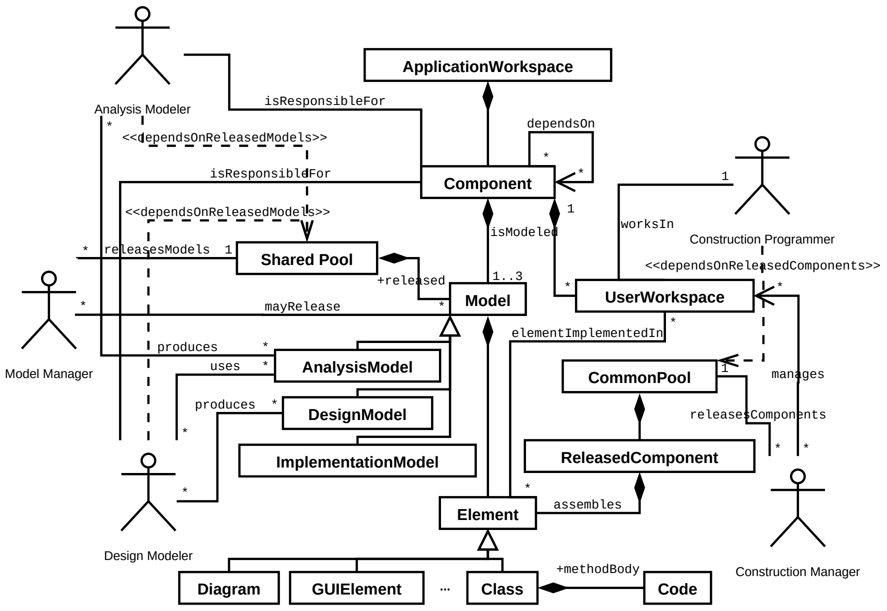

Fig. 16: MasterCraft: class diagram of process-oriented concepts.

4.1 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. Fig. 16 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) focusing on architecture, MasterCraft is oriented towards organizing the development activities in the individual components. 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.

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.

4.2 Developing software with support of MasterCraft and rCOS

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. We describe the support of rCOS with respects to the different roles and their tasks and activities in the project.

Support to administrator 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.

The administrator starts by creating the components identified as groups of related use cases, such as $<< SalesHandler >>$ for use case **UC1**, $<< Inventory >>$ for use cases **UC2**-**UC5**, and component $<< EnterPrise >>$ for the use cases related the the whole enterprise management. 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 $<< SalesHandler >>$ and $<< Inventory >>$, and Brian may be granted these roles for the $<< EnterPrise >>$ component. Furthermore, we have *Martin* who is assigned the global role of Model Manager.

Support to analysis modeler An *Analysis Modeler* starts work on a component by studying its textual requirements. Based on the textual requirements, the Analysis Modeler creates a model of the requirements for the model. This model consists of conceptual class diagrams, use case diagrams, and behavioral models, i.e. the use case sequence diagrams and state diagrams, of the use cases, the specification of the contracts of the use case handlers. For example, Alice has to create the models in Figs. 6-9 and the rCOS specification of the contract. 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. Prototyping can be done and JML-based run-time checking can be applied in addition to the analysis of the requirements specification outlined in Section 3.2.

Note that for formal analysis and its automated tool support, MasterCraft must be extended by adding translators of the UML diagrams into machine readable textual specifications in rCOS. Formal verification and validation tools, such as FDR, SPIN and JML or static checkers must be integrated into MasterCraft so that these tools can be invoked by the analysis modeller. For this, programs for converting rCOS specification to inputs of the tools should be implemented.

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. The release is not to simply drop the model there. The Model Manager should check on the consistency of the model with the others by removing redundancy and integrating identical modelling elements. 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.

Support to design modeller A *Design Modeler* (e.g. Alice) fetches from the shared pool the released model of requirements of a component ($<< SalesHandler >>$ resp.) assigned to her, and refines the analysis model into a logical design model. This involves the application of the expert pattern for refining the use case sequence diagram to a design sequence diagram. The conceptual classes from the analysis model are also refined into design classes.

Then the Design Modeler decompose a component into composition of internal components, and compose a number of components together to for a component model. For example, the original design of $<< SalesHandler >>$ is designed into the composition of $<< SalesHandler >>$, $<< Clock >>$ and $<< Bank >>$ and mark the later two as components already implemented. The Design modeller may also decide on which objects should be persistent, and defines database mapping and primary keys for these classes. This is the case for the decomposition of the $<< Inventory >>$ component into the three layer architecture in Fig. 12. Further, the Design Modeler defines the *component interface* in terms of class operations and queries provided, and may declare additional dependencies on other components. This is case for Alice. She has to declare that component $<< SalesHandler >>$ requires services from $<< Inventory >>$ to get product description and to log a completed sales via interface *CashDeskIf*. It is the same for $<< Inventory >>$ that requires services from $<< Supplier >>$ via interface

SupplierIF. 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.

The design modeller then transform the logical design to a detailed by further refinement rules and patterns, such as class decomposition, refactoring, data encapsulation and synchronization on access to persistent data, and selection of appropriate middlewares replace the object-oriented interfaces in the logical design.

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.

To have formal support from rCOS, MasterCraft should be extended with model transformations that automate the design patterns and other refinement and refactoring rules. Application of refinement rules and design patterns are often constrained by conditions on the models before and after after the transformation. Tools for checking these conditions on the models should be integrated into the MasterCraft environment too. In the current version, MasterCraft has automated transformations to generate code for database quires and for synchronization control on access to shared data. We are now working on QVT implementations of the expert patter, data encapsulation and transformation of an object-oriented model to a component-based model.

Support in 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 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.

Therefore, 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 the model of detailed design defined in rCOS, we can enhance 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 model of the detailed design specifies class invariants *and the functionalities of each class method in terms of its precondition and postcondition. This makes it possible for these conditions to be automatically inserted as assertions into the coded generated.*

Therefore, 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 and JML. 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 is would be a significant for Spect# 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.

5 Conclusion

We have presented our experience in the application of of a formal calculus to the CoCoME case study.

Our experience shows the need of a semantic model that formalises the main concepts and software entities in a model-driven development, and supports multi-view modelling and separation of concerns in a complex software development. rCOS provides these formalisations and support. Model-driven development must also complimented with properties driven analysis techniques. Properties are specified in rCOS as logical formulas and algebraic properties of modelling elements that are formulated as mathematical structures. The algebraic properties form the foundation for model transformations. To ensure consistency and correctness, both static and dynamic consistency of the specification must be checked, and both abstraction and refinement techniques are needed for model transformation and analysis. The work also shows that different models and tools are more effective on the design and analysis of some aspects than the others. Proved correct model transformations should be carried out side by side with verification and validation. Correct model transformations preserve properties that is not required to verify again and verification and validation are used to check the condition on when the transformations can be applied and extra properties required for the transformed model. rCOS is a methodology supports a consistent use of different techniques and tools for modelling, design, verification and validation.

We have analyzed the software development process in a commercially successful tool (MasterCraft [31]) and identified where formal methods support can be "plugged" into the tool to make software development more efficient. However, as discussed in Section 4, there is still a lot to implement to make the tool powerful enough to support the proposed rCOS methodology effectively, and this is part of my current and future work. This is challenging. Yet, our discussion shows that this is feasible. For instance:

1. With the QVT engine that is being developed at TRDDC, we can program the expert pattern, the rule for data encapsulation and structural refinement rules that have proved for rCOS in [23].

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. With human interaction, transformations for decomposing components and composing components in the design stage can be automated.

The current version of MasterCraft does not support the design of controllers of hardware devices and their integration with the application software components and GUI components. However, the discussion at the end of Section 3 shows, this is purely even-based and can be done by following the techniques of embedded systems modelling, design and verification.

Acknowledgements This paper is dedicated to the 70th birthdays of Professors Dines Bjørner and Zhou Chaochen who together founded UNU-IIST in 1992. Also under the supervision of professor Zhou Chaochen, Zhiming Liu did his master study and Xiaoshan Li his PhD degree. The work of Dines on domain engineering and the work of Zhou on logics and calculi have great influence on the research presented in this paper. We would also like to thanks our colleagues who have made contribution in the development of rCOS and CoCoME. They are He Jifeng, Chen Xin, Zhao Liang, Liu Xiaojian, Zhan Naijun, Ander Ravn and Joseph Okika.

References

1. Apache tuscany project. http://incubator.apache.org/tuscany/.
2. The concurrency workbench. http://homepages.inf.ed.ac.uk/perdita/cwb//.
3. Service component architecture. http://www.osoa.org/display/Main/Home.
4. Modelling contest: Common component modelling example (CoCoME). http://agrausch.informatik.uni-kl.de/CoCoME, 2007.
5. R.J.R. Back and J. von Eright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science, Springer-Verlag, 1998.
6. K. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
7. X. Chen, J. He, Z. Liu, and N. Zhan. A model of component-based programing. In *Proc. International Symposium on Fundamentals of Software Engineering, to appear in LNCS*. Springer, 2007.
8. 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.
9. Z. Chen, Z. Liu, V. Stolz, L. Yang, and A.P. Ravn. A Refinement Driven Component-Based Design. In *Proc. of 12th IEEE Intl. Conf. on Engineering of Complex Computer Systems (ICECCS 07), to appear*. IEEE, 2007.
10. Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors. *Formal Methods for Components and Objects, Second International Symposium, FMCO 2003, Leiden, The Netherlands, November 4-7, 2003, Revised Lectures*, volume 3188 of *Lecture Notes in Computer Science*. Springer, 2004.
11. C. Flanagan, *et al.* Extended Static Checking for Java. In *Pro. PLDI' 2002*, 2002.

12. M. Fowler, *et al.* *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
13. J. He, X. Li, and Z. Liu. Component-Based Software Engineering. In *Pro. ICTAC'2005, Lecture Notes in Computer Science 3722.* Springer, 2005.
14. J. He, X. Li, and Z. Liu. rCOS: A refinement calculus for object systems. *Theoretical Computer Science*, 365(1-2):109–142, 2006.
15. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
16. C.A.R. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.
17. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.
18. P. Kruchten. *The Rational Unified Process – An Introduction.* Addison-Wesly, 2000.
19. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
20. C. Larman. *Applying UML and Patterns.* Prentice-Hall International, 2001.
21. Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.
22. J.L. Leavens. JML's rich, inherited specification for behavioural subtypes. In *Proc. 8th International Conference on Formal Engineering Methods (ICFEM06)*, volume 4260 of *LNCS.* Springer, 2006.
23. X. Li, Z. Liu, and L. Zhao. Object-oriented structure refinement - a graph transformational approach. Technical Report 340, UNU-IIST, P.O. Box 3058, Macao SAR, China, 2006. Published in Proc. International Workshop on Refinement, ENTCS, and extended version is accepted by Formal Aspect of Computing.
24. Z. Liu. A continuous algebraic semantics of CSP. *Journal of Computer Science and Technology*, 4(4):304–314, 1989.
25. Z. Liu and J. He (Eds.). *Mathematical Frameworks for Component software: Models for Analysis and Synthesis, Series on Component-Based Software Development - Vol. 2.* World Scientific, 2006.
26. Z. Liu, A.P. Ravn, and X. Li. Unifying proof methodologies of duration calculus and timed linear temporal logic. *Formal Aspects of Computing*, 16(2):140–154, 2004.
27. Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2003.
28. B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.
29. R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag,, 1980.
30. A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.
31. Tata Consultancy Services. MasterCraft. http://www.tata-mastercraft.com/.
32. C.C. Zhou, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269276, 1991.