



The United Nations
University

UNU-IIST

International Institute for
Software Technology

rCOS: Theory and Tool for Component-Based Model Driven Development

Zhiming Liu, Charles Morisset and Volker Stolz

February 2009

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



The United Nations
University

UNU-IIST

**International Institute for
Software Technology**

P.O. Box 3058
Macao

rCOS: Theory and Tool for Component-Based Model Driven Development

Zhiming Liu, Charles Morisset and Volker Stolz

Abstract

We present the roadmap of the design and progress of a theory supported tool for component-based model driven software development (CB-MDD). First the motivation for using CB-MDD, its needs for a theoretical foundation and tool support are discussed, followed by an overview of the development of the theory and a prototype tool. The initial experiences with the tool are summarized, and based on the lessons learned, the further development trajectory leading to further integration with transformation and analysis plug-ins is delineated.

Keywords: *contract, component, design pattern, model transformation*

This paper is based on the keynote to be delivered by Zhiming Liu at FSEN 2009 and will be published in the proceedings of the conference by LNCS. The research supported projects HighQSoftD and HTTS funded by Macau Science and Technology Development Fund.

Zhiming Liu is a Senior Research Fellow at UNU-IIST. His research interests include theory of computing systems, emphasising sound methods and tools for specification, verification and refinement of fault-tolerant, realtime and concurrent systems, and formal techniques for OO development. Email: Z.Liu@iist.unu.edu

Charles Morisset is a postdoctoral research fellow of UNU-IIST working on the projects HighQSoftD and HTTS funded by Macao Science and Technology Development Fund. His research interest include computer security, formal verification and rCOS tool development. Email: morisset@iist.unu.edu

Volker Stolz is an assistant research fellow of UNU-IIST working on the projects HighQSoftD and HTTS funded by Macao Science and Technology Development Fund. In particular he leading the rCOS tool development project. His research interest also includes runtime verification and model driven software development. Email: vs@iist.unu.edu

Contents

1	Introduction	1
2	Natural Path to CB-MDD	2
2.1	Early notions of components and models	2
2.2	Theoretical and tool support to successful CB-MDD	3
3	Theoretical Foundation of rCOS	5
3.1	Component implementation and component refinement	5
3.2	Contracts	7
3.3	Composition	9
4	The rCOS Tool	10
4.1	Tool support to requirement analysis	10
4.2	Model transformation tool to support design	12
5	Concluding Remarks	14

1 Introduction

Complexity has long been recognized as an *essential property* of software, not an *accidental one* [13, 14]. The inherent complexity is due to four fundamental attributes; the *complexity of the domain application*, the *difficulty of managing the development process*, the *flexibility possible* to offer through software, and the *problem of characterizing the behavior* of software systems [2]. The first three attributes focus on the problem of *changeability* of software to meet continuously changing requirements for additional functionality and features, and the final one pin-points the difficulty in software *analysis, validation and verification* for *correctness assurance*.

We are now facing an even greater scale of complexity with modern *software-intensive systems* [41]. We see these systems in our everyday life, such as in aircraft, cars, banks and supermarkets [6]. These systems provide their users with a large variety of *services* and *features*. They are becoming increasingly *distributed, dynamic* and *mobile*. Their components are *deployed* over large networks of *heterogeneous platforms* and thus the *interoperability* of the distributed components becomes important. Components are also *embedded* within hardware devices. In addition to the complexity of functional structure and behavior, modern software systems have complex aspects concerning *organizational structure* (i.e. *system topology*), *distribution, interactions, security* and *real-time*.

A complex system is open to total breakdown [35], and we suffer from the long lasting *software crisis*¹ where projects fail due to our failure to master the complexity. Given that the global economy, as well as our every day life, depends on software systems, we cannot give up in advancing the theory and the engineering methods to master the increasing complexity of software development.

In this paper, we present the rCOS approach to CB-MDD for overcoming software complexity. Section 2 motivates the research in the rCOS modeling theory and the development of a prototype tool. There, we first show how CB-MDD is a natural path in the advance of software engineering. It is then followed by a discussion of the key theme, principles, challenges, and essential techniques of a component-based model driven approach. We stress the importance and the add-ins of *tool support*, and argue that a tool must be supported by a sound and appropriate *modeling theory*. In Section 3, we summarize the theoretical aspects of rCOS and show how they meet the needs discussed in Section 2. Section 4 reports our initial experience with the tool, and based on the lessons learned, we delineate the further development trajectory leading to further integration with transformation and analysis plug-ins. Concluding remarks are given in Section 5.

¹Booch even calls this state of affairs “normal” [2].

2 Natural Path to CB-MDD

There is no other better way of mastering a complex system development than *separation of concerns*. Within the scope of software engineering, models of software development processes are proposed for dividing and conquering the problems in software development. The traditional approaches are mostly variants of the *waterfall model* in that problems of software development are divided into the problems of *requirements capture and analysis, design, coding and testing*, and solved at different level of abstractions.

2.1 Early notions of components and models

In a waterfall process, the principles of *structured programming* [11] and *modularization* [34] are used to construct a software system by decomposing it into *procedures* and *modules*. The concept of modules is thus an early analogy of the notion of *components*. With the notions of decomposition and modules, the initial waterfall process changed into the *evolutionary development*, and the *spiral model* is proposed with more consideration of project management and risks control [38]. Procedures as components do not support large scale software development, and the original modules as components do not have *explicitly specified contracts of interfaces* to be used in third party composition [40].

For applications with higher demands for correctness assurance and dependability, rigorous testing after implementation for *software defect detection* is not enough, and a best effort at defect detection is required in each phase of the development process. This advances the waterfall model to the *V-model*. In a V-model development process, *artifacts* produced in each phase should be properly documented and validated. *Tools* are then developed to help in the documentation of the artifacts produced in different cycles for different versions to ensure their consistency. Tools for prototyping and simulation are also used for system validation in different phases. Taking documents of artifacts as “models”, though not necessarily formal models, a non-trivial software development is to produce, validate, and test models with some tool support.

Software systems for safety critical applications are required to be *provably correct*. For this, each phase is required to produce *verifiable models* and this needs the *modeling notation to be formally defined* and a *sound theory* to be developed for verifying and reasoning about properties of models. Model verification is only realistic with support of automated tools. Indeed, in the last forty years or so, a large body of formal notations and theories have been developed based on two different models, *state based* [22, 42, 37] and *event based* [19, 29]. Development of *theorem proving* [39], *model checking* [21, 36] and *simulation* [10] tools have rapidly advanced recently. All of these methods and techniques have their uses in some aspects of system development, and the challenge is now to select and adapt them to a harmonic whole.

2.2 Theoretical and tool support to successful CB-MDD

Components and models are in the scope of software engineering. In industrial practice models are often manually built, and an initial code outline is generated from a model of a detailed design. Only recently component-based model driven design is becoming a clear and mainstream discipline. The discipline requires that in a development process

- each phase is based on the *construction of models*²,
- models in later phases are constructed from those in earlier phases by *model transformations*,
- code is an executable model generated from models in the design phase.

This implies that what is critical to CB-MDD is a modeling approach with sound *theoretical foundation* and strong *technical and tool support*. In what follows we describe the key features of this modeling approach.

Multi-dimension separation of concerns The models at each phase should separate the *different views* and characterize the different aspects of the software *architecture* at the given level of abstraction. A unified set of different notations, such as UML [31], is often used as the modeling language. There is also a consensus on notations for the different views of a software system, such as use cases for requirements, class diagrams for structural design, sequence and state diagrams for interaction protocols and reactive behaviors. Separation of concerns³ has to deal with the important issue of the *correctness* and *consistency* of *different views* [5, 33]. There are existing UML profiles with precisely defined syntaxes and tools for building models, which ensure and check their syntactic correctness and consistency. The problem of semantic correctness and consistency is much harder, and a semantic theory has to be developed for the unified modeling language. Our experience with rCOS [4, 7] is that the challenge lies in that the semantic theory must support separation of concerns to allow us to factor the system model into models of different views and to consistently integrate models together under an execution semantics of the whole system. This is even more difficult when we have to integrate *object-orientation* into a theory of CB-MDD [18]. The semantic theory is also needed for the models to be verifiable by verification tools and *manipulatable* by automated correctness preserving model transformations, and properly formalized models are needed for automatic generation of test suites. Separation of concerns and multi-view modeling allow us to take advantage of different theories and their tool support for the analysis and manipulation of different views and aspects. To this end, we need to advance the ideas of “putting theories together” [3, 16] and “unifying semantic theories of programming” [20] to produce, analyze and transform models with integrated tool support.

²For a safety critical application, these models should be verifiable.

³Here, multi-dimensional separation of concerns is related to what it meant in [32], but with a wider extension.

Object-orientation in CB-MDD Why do we need object-orientation? Among other reasons including reusability and maintenance, we give three reasons from our own experience. Firstly, object-oriented analysis and design complements structured analysis and design in handling the complexity of the organizational structure of the system [2]. *Concepts*, their *instances* and their *relations* in the application domain naturally form the *structure of the domain* and can be directly modeled by the notions of *classes*, *objects* and *associations* or *attributes*. The combination of use cases for functional requirements analysis and decomposition with object orientation for structural analysis and decomposition works systematically and effectively in both practical and formal component-based development [24, 7]. Secondly, the use of design patterns [15, 24] makes the object-oriented design through model transformations more systematic and thus has much higher possibility for automation [18, 28]. Finally, most, if not all, industrial component-based technologies are implemented in the object-oriented paradigm.

Component-based architecture For a model driven design, we need a precise and strong notion of *component-based architecture* such that

1. it describes the system functionalities and behavior,
2. it captures the decomposition of the system into *components* that *cooperate* and *interact* to fulfill the overall system functionalities,
3. it supports *composition*, *coordination* and *connection* of components and describes the *hierarchical* and *dependency* relationships among *components*.

Unlike the conventional notions of components which used to be about programs in code, components in CB-MMD are involved in all phases of the development and represented in different languages. Composition, coordination and connection of components can only be defined based on the *interface behaviors* of the components. Independent deployment, interoperability, reuse of existing (commercial off-the-shelf) components also require components to have explicitly specified *contracts of interfaces* [40, 4, 7]. The modeling language should be expressive enough for specifying the multi-view and hierarchical nature of components and allows *abstraction by information hiding*.

Scaling and automating refinement Formal techniques and tools for verification are mainly for defect detection. They do not support decision making of the designer in systematic and correct model construction. The basic notation we find in formal methods that supports correct by design is *program refinement* [30, 1]. However, the classical refinement techniques need to be generalized and unified to support the separation of concerns in the multi-view modeling paradigm and allow models of different views to be refined separately and hierarchically, such as data *functionality refinement*, *interaction refinement* and *object-oriented structure refinement* [18, 4, 44]. Refinement rules need to be scaled up and automated via exploration of formal use [18, 28] and automation of design patterns [15, 24] for abstract models and refactoring rules [12]

for models at lower levels of abstraction, such as a design class diagram. Automation of the scaled refinement rules provides us with the implementation of model transformations. Application of such a model transformation generates *proof obligations* for verification of properties the models before and after the transformation. These verification tasks can be carried out by an integrated verification tool, such as a theorem prover, a static checker, or a model checker. This gives a seamless combination of tools for verification and correctness by construction. Therefore, automation of a rich set of refinement rules is the key to extensively tool supported model driven development. It turns out to be the greatest challenge, too, in the tool development.

Tool supported development process A convincing conclusion drawn from the above discussion is that CB-MMD needs an integrated tool suite supported by a sound theory, instead of a single purpose tool. With the tool support, a software system must be developed in a clearly defined engineering process in which different activities at different stages of development are performed by project participants in different roles. We see this very important, as it allows us to define at which points 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.

3 Theoretical Foundation of rCOS

We summarize the main concepts and models of software artifacts defined in rCOS without going into technical details. Such details can be found in our earlier publications [18, 4, 7, 44].

An essential concept in rCOS is that of *components*. A component K has various models with different details in different stages and for different purposes. A *component implementation* as a piece of program, *contracts* of components at the level of requirements specification and design, and *publications* for component usages and synthesis.

3.1 Component implementation and component refinement

At the source code level, components has a *provided interface* $K.pIF$, possibly a *required interface* $K.rIF$ and a piece of program code $K.code(m)$ for each method $m()$ in the provided interface. The required interface $K.rIF$ contains the signatures of methods that are called in the code of the component K .

Interfaces In rCOS, an interface is a *syntactic notion*, and each *interface* I is a declaration of a set $I.fields$ of typed variables of the form $x : T$, called *fields*, and a set $I.methods$ of method signatures of the form $m(x : T, y : V)$, where $x : T$ and $y : V$ are the input and output parameters with their types.

UTP as root of semantic theory In principle, different components can be implemented in different programming languages. We thus need a semantic model for “unifying theories” of different programming languages, and thanks to Hoare and He, we use the well studied UTP [20]. The essential theme of UTP that helps rCOS is that *a program in any programming language can be defined as a predicate*, called a *design*. A *design* is specified as a pair of pre- and post-conditions, denoted as $pre(x) \vdash post(x, x')$, of the *observables* x and x' of the program, and it says that if the program executes from a state where the *initial values* x satisfies $pre(x)$ the programs will terminate in a state where the final values x' satisfies the relation $post(x, x')$ with the initial values x . Observables include program variables and auxiliary variables dependable on the observable behavior being defined, such as termination, denoted by ok and ok' in sequential programs and interaction traces tr and tr' in communicating programs.

Semantics and refinement of components With the definition of designs and the calculus established in UTP, the semantics of a component K is defined as a function $\lambda C_{rIF} \cdot spc.K$ such that for any function C , called a *contract* of the required interface $K.rIF$, which gives each required method $n()$ a design $C(n())$, $spc.K(C)$ defines the semantics of each $m()$ of the provided interface $K.pIF$ as a design. The semantics $spc.K(C)(m)$ is calculated from the code of m by replacing each invocation to a required method $n()$ by the semantics $C(n())$. Notice that if $K.rIF$ is an empty interface, $\lambda C_{rIF} \cdot spc.K$ is a constant function.

Components are in general reactive programs and thus concurrent programming languages are used for their implementation. The semantics of each method is thus defined as a *reactive design*. In [4], the *domain of reactive designs* \mathcal{RD} is a sub-domain of the domain of designs \mathcal{D} characterized by *lifting function* $\mathcal{H} : \mathcal{D} \rightarrow \mathcal{RD}$ such that $\mathcal{H}(p \vdash R) \hat{=} (true \vdash wait') \triangleleft wait \triangleright (p \vdash R)$. In this specification, Boolean observables $wait$ and $wait'$ represents the synchronization so that the execution of the program is suspended when it is in a *wait* state. We also introduce *guarded designs* $g \& (p \vdash R)$ to specify the reactive behavior $\mathcal{H}(p \vdash R) \triangleleft g \triangleright (true \vdash wait')$.

A component K_1 is a *refinement* of K if they have the same provided and required interfaces, and for each contract C of $K.rIF$, and each provided method $m \in K.pIF$, the design $spc.K_1(C)(m)$ is a refinement of the design $spc.K(C)(m)$ in the refinement calculus of UTP.

Object-orientation in rCOS To support object-oriented design of components, types of fields of component interfaces can be *classes* and thus the values of these fields are objects. We have extended UTP to define object-oriented programs and developed an object-oriented refinement calculus to handle both structure and behavior refinement [18, 44]. The object-oriented semantic model in rCOS provides formal treatment of *aliasing*, *inheritance* and *dynamic binding*. These features are needed in CB-MDD when constructing, transforming and verifying models in later stages of the development.

3.2 Contracts

In the CB-MDD paradigm, a component is developed by model transformations from its *requirements analysis model* to a *design model* and finally an *implementation model*. We take the view that the analysis model specifies the functionalities from the users' perspective and describes *what* does the component do for *what kind* of users. However, a user does not have to be human. A kind of users is called an *actor* in the UML community and the actors together define the *environment* of the component.

Component-based development allows the use of an existing component to realize a model of a component in the analysis model. The *fitness* of the existing component for the purpose in the model must be checkable without information about the design and implementation of the existing component. For this, the analysis model of a component should be a black box characterization of what is needed for the component to be designed and used in building and maintaining a software. The information needed depends on the application of the component. For example, for a sequential program, the specification of the static data functionality of the interface methods is enough, but information about the worst execution time of methods is needed for a real-time application, and for reactive systems we also need reactive behavior of the component and the interaction protocol in which the environment interacts with the component. For the treatment of different applications, the intention of rCOS is to support incremental modeling and separation of concern.

Contracts for multi-view modeling In rCOS, a black box behavior model of interfaces called a *contract* is defined, and its current version focuses on reactive systems. A contract $C = (I, \theta, \mathcal{S}, \mathcal{T})$ defines for an interface I , denoted by $C.IF$,

- an *initial condition* θ , denoted by $C.init$, that specifies the allowable initial states of the intended component,
- a specification function \mathcal{S} , denoted by $C.spec$, specifying the reactive behavior by giving each method $m \in C.IF.methods$ a reactive design $C.spec(m)$, and
- a protocol \mathcal{T} , denoted by $C.prot$, that a set of finite traces over the method names $C.IF.methods$, specifying the assumed *work flows* or *interaction protocol* in which the actors use services of the intended component.

Note that the specification function $C.spec$ combines the static (data) functionality view and the reactive dynamic behavior view [5]. The data functionality view is modeled by *non-reactive designs* and the reactive behavior by a UML *state diagram*. A contract has to be *consistent* such that if the actors follow the protocol in their work flow they should not be blocked by the component. The *interaction view* is represented as a UML *sequence diagram*. A contract also has the fourth view, the *structure view*, that defines the data- and class structures. It is represented by a UML *class diagram*.

Refinement of contracts For the study of the consistency of contracts, separation of concerns and refinement among contracts, an *execution semantics* of a contract C (and thus components) is defined in [4] by its *failures*, $failure.C$, and its *divergences*, $divergence.C_2$ [36]. A contract C_1 *refines* a contract C_2 , denoted by $C_2 \sqsubseteq C_1$, if C_1 is neither more likely to diverge, i.e. $divergence.C_1 \subseteq divergence.C_2$, nor more likely to block the actors, i.e. $failure.C_1 \subseteq failure.C_2$.

Theorem of separation of concerns The theorem of separation of concerns in [4] allows to refine the static functionality and reactive behavior separately to preserve the consistency. It is interesting to point out that object-oriented refinement [18, 44] makes formal use of design patterns. It is thus crucially important for the refinement of the static functionality and for scaling and automating refinement to develop tool support.

Correctness and publication of components A component K *fulfills* or *implements* a contract C if there exists a contract such that $C \sqsubseteq spec.K(C_r)$. Thus, it is now clear that $\lambda C_r IF \cdot spec.K$ calculates a contract of the provided interface of K for a given contract of C_r of its required interface, and $spec.K(C_r)$ is the strongest **provided contract** for C_r . We call a pair of contracts $C = (P, R)$ of $K.pIF$ and $K.rIF$ a *contract of component* K if $P \sqsubseteq spec.K(R)$, that is K fulfills the provided services P if the environment provides K with services R .

The notion of contracts of interfaces is closely related to the notion of *component publications* [17, 23] for assembling components. A *publication* of a component is a specification $U = (G, A)$, where G and A are contracts of the provided interface $K.pIF$ and required interface $K.rIF$, respectively, but where $spec.G$ and $spec.A$ only specify methods with non-reactive designs. Therefore, a publication of K is an abstraction of a contract of K . In [23], a function \mathcal{P} is defined to obtain a publication from a contract $C = (P, R)$ of K . \mathcal{P} maps each reactive design D to a *pure functionality design* $\mathcal{P}(D) \hat{=} D[false/wait, false/wait']$, by substituting the Boolean variables $wait$ and $wait'$ with $false$. A publication $U = (G, A)$ of K is *faithful* if there is a contract $C = (P, R)$ of K such that $U \sqsubseteq \mathcal{P}(C)$, i.e. $G \sqsubseteq \mathcal{P}(P)$ and $A \sqsupseteq \mathcal{P}(R)$. This is the basis of *publication certification*. A component K with a faithful publication U *fits* in the position of contract $C = (P, R)$ in a model, if U refines $\mathcal{P}(C)$. In [23], a mapping \mathcal{C} from publications to contracts is also defined and a theorem is proven that $(\mathcal{P}, \mathcal{C})$ forms a *Galois connection* between the domain of contracts and the domain of publications.

A component K with a publication $U = (G, A)$ is *substitutable* by a component K_1 with a publication $U_1 = (G_1, A_1)$ if $U \sqsubseteq U_1$ that is defined as $G \sqsubseteq G_1$ and $A \sqsupseteq A_1$. Notice that contracts and publications of a component are truly *black box specifications* of the component. Contracts are used for correct design and correctness verification while publications are used for substitutability and assembling.

3.3 Composition

The notion of *composition* is essential for a component-based design and must be defined for models of components at all the levels of abstraction, and it should be consistently refactorable to composition of interfaces, static functionality, reactive behaviors and interaction protocols. In rCOS, we define the basic composition operators for *renaming* interface methods, *restriction* on the environment from access to provided methods, *internalization* of provided methods to make them autonomously executed when they are enabled, *plugging* the provided interface of one component to the required interface of another components, *disjoint parallel composition* of components, and *synchronization* of provided methods to coordinating them. In the following, we discuss the nature of these composition operators at different levels of abstraction.

Composition of contracts and publications of components The definitions of *renaming*, *restriction*, *plugging* and *disjoint parallel compositions* for contracts and publications are relatively easier than those for *internalization* and *coordination* [4]. However, for the plugging composition, the *composability condition*. A contract $C_1 = (P_1, R_1)$ (or a publication $P_1 = (G_1, A_1)$) is composable with $C_2 = (P_2, R_2)$ (resp. publication $P_2 = (G_2, A_2)$) if the provided contract P_1 in C_1 (resp. G_1 in P_1) refines the required contract R_2 in C_2 (resp. A_2 in P_2).

The difficulty in defining internalization and synchronization is first to make sure the result of a composition is still a contract defined in rCOS. For this, the effects of the autonomous internal executions of the internalized or the synchronized methods must be aggregated into the remaining interface methods. In [4] a definition for synchronization of a component by a *process* is given and the result proven to be a component. However, this is better at used at the implementation level. In [23], the details are given for internalization, and in a similar way we can define a synchronization of operation on a component.

Composition of component implementations The composition operators *renaming*, *restriction*, *plugging* and *disjoint parallel composition* are implemented as simple *connectors* following the semantics defined in [4, 23].

Internalization and synchronization are implemented by using *processes* (think of a scheduling process) that automatically calls the internalized methods and the synchronized methods respectively for execution when they are enabled (i.e. their guards become true). The semantics of the synchronous composition of a component and a process is defined in [4] and there the composed entity is proven to be a component.

Compositional modeling, refinement and verification At all levels of abstraction, the composition operations are proven to be monotonic with respect to the refinement order. This allows us to carry out compositional design by model transformations. The relationships of fulfillment of contracts by components, faithfulness of component publications, and the fitness

of a component in a model are preserved by the composition operations (for composable compositions). This enables us to do compositional analysis, verification and certification.

4 The rCOS Tool

The rCOS tool focuses on CB-MDD and is oriented towards organizing the development activities. It introduces a body of concepts and a hierarchy of artifact repositories, designed to support team collaboration on development of the models and generation of code (cf. the paragraph on **tool supported development process** in Section 2.2). 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 partitioned into *components* through hierarchical use cases. A component is characterized by its subset in the model of different views and represented in different forms depending on the phases of the development. The application maintains the (*requirements*) *analysis model*, the *design model* and the *platform specific design and implementation*. The informal requirements document is mainly a description of the use cases and their relationships. It is represented in a structured natural language (not stored in the model) and the use case diagrams [26, 7]. Use cases may *refer* or *use* to other use cases, hence the hierarchical notion of sub-use cases. *Each use case is called a component at this level.*

To support construction, understanding and transformations of these models, a *UML profile* is defined for rCOS, and models of the views of reactive behavior, interaction and class structure are created as instances of the metamodel of the UML profile [8]. The UML model has an equivalent representation in the rCOS textual syntax and is the input for the various formal analyses.

4.1 Tool support to requirement analysis

An *Analyst* works on a component, that is, a use case of the application, by studying its textual requirements and the application domain, and creates an *analysis model* consisting of a *use case diagram*, a *conceptual class diagram*, an *interface sequence diagram*, the *functionality specification* of the interface methods, and a *state machine diagram*. The use case diagram represents the dependency relation between the actors in this use case and referenced use cases.

Models of different views The use case diagram describes the dependency relationships between the actors and the component. Some actors are components external to this component. The conceptual class model represent the domain concepts and objects with their structural inheritance relations involved in the use cases of the use case diagram. Methods are designed for the component interface, but not for the other conceptual classes at this stage. The interface sequence diagram models the interactions between the actors and the component, and the interactions among the subcomponents corresponding to the sub-use cases of the use case diagram.

It is thus in general a *component sequence diagram* (cf. the next subsection). The state diagram represents reactive behavior of the component and characterizes the flow of control and synchronization of the component. The functionality specification of the interface methods specifies the pre- and post-conditions of the methods in rCOS.

Analysis and validation The Analyst is responsible for verifying that the models of the different views are consistent, and validating it against the informal requirements document. The syntactical consistency checking is implemented as part of the type checker of rCOS, though the full implementation is still ongoing.

We have developed a prototype of a tool for automatic prototyping from an analysis model [25] for validating requirements. For the dynamic consistency of the sequence diagram and state machine diagram, we translate them into CSP processes and check *deadlock freedom* of their composition with the CSP model checker FDR2 [9]. Here, consistency means that all interaction scenarios defined by the sequence diagram are accepted by the state machine. Likewise, we check *faithfulness of the contract* with regard to an executable rCOS specification (the component does not deadlock for any interaction in the contract).

Application dependent properties, such as safety and liveness, can be verified by a combination of model checking the CSP process of the state diagram and static analysis of the functionality specification of the interface methods.

The Analyst may iterate over this model, creating, decomposing and refining models. It may also be necessary to revise the informal requirements documents according to the results of the analysis and validation. She can declare a dependency on another component and, if the component depends on other components, the Analyst specifies which interface these *required components* have to provide, or she may introduce abstract models of the required components. A verified and validated model can be *frozen* and is used for the design of components by a *Designer*.

Remarks With the support of the tool, the syntactic consistency can be guaranteed, such as the method names, parameters and name of attributes in different views. The construction of the class diagram, sequence diagram and state diagram is fully supported by the tool. It however needs domain experts who understand the syntax and its semantics for writing the correct pre- and post-conditions of the methods. Another difficulty arises in a team where with multiple analysts working on different components (even initially disjoint components). Decomposition of a use case component requires the awareness of the progress being made on other components to avoid duplicate introduction of components and to accommodate changes obtained through analysis and design of other components. There seems to be no formal and systematic tool support to ensure across component consistency except for having project review meetings to decide what changes should be made. Our experience from the case study is that modelers of different components have to spend a lot time to discuss with each other the models that they are working on and informing each other about any new components they introduce.

4.2 Model transformation tool to support design

A *Designer* produces a *design model* from an analyzed component model by a sequence of model transforms. In rCOS, we intend to support three kinds of model transformations for producing an *object-oriented design model* of the component, a *component-based design model* from the object-oriented design model, and a *platform specific design model*.

Object-oriented design of a component This mainly involves stepwise refinement of the data functionality of the interface methods. The driving force for this is repeated applications of the *Expert Pattern* for Assignment of Responsibilities in object-oriented design [24]. The *Expert Pattern* provides a systematic decomposition of the functionality of a method of an object into responsibilities of its related objects, called *information experts*, which *maintain* or *know* the information for carrying out these responsibilities. The related objects of an object o can be defined by the navigation paths from o , and they are derivable from the class diagram (and from the rCOS class declarations).

Formalizing and implementing Expert Pattern We classify the primitive responsibilities of an object $M :: o$ of class M into *knowing responsibilities* and *doing responsibilities* [24]. Each object is considered to be responsible for knowing its attributes and doing its methods. It is also responsible for knowing its linked objects and for delegating tasks to them, i.e. invoking their methods. For instance, if an object a contains an object b , then a can access to fields and methods of b , but if b contains an object c , then a should not access directly to attributes and methods of c , but rather delegate such actions to b .

Hence, we introduce the following rewriting rules, for any navigation path $p \neq \text{this}$ of type M , any fields a and b and any methods m and n :

$$\begin{array}{llll} p.a.b & \longrightarrow & p.find_a.b() & p.a.m(\bar{x}) & \longrightarrow & p.find_a.m(\bar{x}) \\ p.m(\bar{x}).a & \longrightarrow & p.find_m.a(\bar{x}) & p.m(\bar{x}_1).n(\bar{x}_2) & \longrightarrow & p.find_m.n(\bar{x}_1, \bar{x}_2) \end{array}$$

where the following methods are automatically created when needed:

```
M :: find_a_b() {return a.b}
M :: find_a_m(p) {return a.m(p)}
M :: find_m_a(p) {return m(p).a}
M :: find_m_n(p, q) {return m(p).n(q)}
```

These rules can be inductively applied to any navigation path containing at least three elements different from *this*, the number of elements in the path being decreased by one at each step.

What a method of an object can do, is to change its own attribute and to delegate the change of the attributes of its linked objects to the corresponding objects. We also introduce a rule

concerning the responsibility of objects with respect to the modification of their attributes. For any navigation path $p \neq \text{this}$ of type M , any attribute a and any expression e , we introduce the following rule:

$$p.a := e \longrightarrow p.\text{set}_a(e) \text{ with } M :: \text{set}_a(x) \{a := x\}$$

At the moment, we have implemented a transformation which takes the full specification of a method $m()$ in normal form and carries out all the responsibility assignments in one go. We plan to implement a transformation that takes a part of a specification designated by the user and carries out one step of the decomposition. For example, she selects a message in a sequence diagram, chooses a sub-expression from the corresponding functionality specification and asks the system to generate the intermediate setters and getters to delegate the accesses. This also generates a more readable design, because it allows the designer to choose meaningful method names.

Other model transformations Before and after the application of the expert pattern, we can improve the low level design by using other design patterns, such a *High Cohesion* and *Low Coupling* (cf. [24] for informal presentation and [18] for an rCOS formalization) as well as the Creator Patterns, Structure Patterns and Behavior Patterns (cf. [15] for informal description and [28] for the rCOS formalization) and refactoring rules (cf. [12] for informal discussion and [28, 44] for the rCOS formalization). Some of these patterns introduce new classes and decompose classes. We plan to implement a library of design patterns and refactoring rules in the rCOS tool. A design pattern or a refactoring rule has conditions on the model before and after the transformation. The application of the corresponding automated transformation generates these conditions as proof obligations to be proved by using theorem proving and/or model checking. This is how verification tools are to be integrated into the rCOS tool, that is, we propose a tool suite for *verification which is integrated into model transformations*.

Effect of transformations The application of these transformations to an analysis model refines the interface sequence diagram to an *object sequence diagram* and the conceptual class diagram to a *design class diagram* in which methods of a class are introduced, and the functionality specification of interface methods into invocations of the newly introduced methods of the classes and *specification statements* of these methods in their classes. Now the design class diagram can be automatically produced for a transformation, but the automatic generation of the object sequence diagram is harder and yet to be automated.

Discussion Pre-processing of the functionality specification of the method is needed so that it is decomposed into specifications in terms of primitive responsibilities. This sounds unrealistic. However, the practical engineering guidance that the precondition of a method is mainly to check conditions on existing objects and the postcondition are mainly about which new objects were created, old object deleted, what attributes modification were made on which existing

objects. Our experience is that with the class diagram this guidance actually helps in writing and understanding the functionality specification of a method in a normal form that is essentially a *conjunction of disjunctions of sequential compositions of primitive responsibilities* [7]. An expression can represent a significant computation, such as the greatest common divisor of two integers or the shortest paths between two nodes of a graph, and it needs to be coded by a *programmer*. We have also defined refinement rules to transform universally and existentially quantified specification statements into loops [7]. These rules can be easily automated.

Component-based design The designer takes the object-oriented design model and identifies “permanent objects” and decides if they should be made into components according to their features. The features include if they logically represents existing components, hardware devices, external subsystems, or they can be reused in different models of this application and other applications. A permanent object that aggregates a large amount objects and functionality is also suggested to be made into a component. The identification of objects as subcomponents in the object sequence diagram also defines the interfaces among the subcomponents. We can then *abstract* the object sequence diagram into a *component sequence diagram* by hiding the object interactions inside the identified components [43]. This step of abstraction is yet to be automated as a transformation. It generates the invariant that none of these objects is *null* for proof that the identified objects are indeed permanent. The execution of this transformation will also produce a *component diagram* representing the original component as the composition of the identified components. Reuse of existing designed components is also decided when applying the transformation. This generates proof obligations for checking fitness and composability of existing design components.

Platform specific design and implementation The Designer decides on components that should maintain persistent data, and defines database mappings and primary keys for these components, and plans database queries to access the persistent data.

The model of component-based design obtained from the object-oriented design services as the *platform independent design* and employs direct object method interactions. The Designer studies the nature of the components, such as their distribution and deployment requirements, and decides the concrete interaction mechanisms and middlewares for individual interfaces.

5 Concluding Remarks

We have presented the motivation, the theme, the features and challenges of the rCOS theory and its tool support. The presentation is mostly informal, but what we are delighted about is that all the informal concepts, artifacts and design activities have their formalized versions in the rCOS theory and formulated in the roadmap of the design of the rCOS tool (cf. [18, 4, 7]). We take this as a promising sign of the research as we believe a theory and a tool can be effective

only when they can be embedded into a practical software engineering processes. Except for application specific significant algorithms, nearly all the code can be automatically generated from a well specified design model. Also, transformations from a platform independent design to a platform specific design with existing industry standards can be mostly automated.

We have left out the discussion about *system integration* of the paper due the lack of space. System integration is mainly about the design of GUI objects and hardware controller that need to interact with each other and with the domain components. Modeling, analysis and design of these interactions can be done in a pure event-based modeling theory and its tools support for embedded information systems design [7].

There is a long way to fulfill our vision on the design and the implementation of the tool set out in [27]. The main challenge is still in the automation of model transformations from analysis models to platform independent design models. It is not enough to only provide a library of implementation transformations, but more importantly, the tool should provide guiding information on which rule is to be used. It is also difficult to support consistent and correct reuse of already designed methods when applying the Expert pattern to design a method, and the reuse of already designed components when designing a new component.

On the engineering side, our tool shows the same aspects and their respective problems as the software engineering discipline we would like to apply it to: development of the tool requires understanding of formal methods to correctly encode the requirements and algorithms, such as transformations and their preconditions, just as the model designers need to understand how to properly model a contract (including technicalities that might be necessary to make a problem actually amenable to the model checked), or write relational functionality specifications. While our progress in the tool development is steady but slow, we feel that it is well in scope of a commercial application from usability, presentation and documentation, included guided storytelling of use cases. As a matter of fact, we have *only* been able to make this progress with our limited resources because we have been harnessing existing infrastructure such as UML for modelling, and the QVT language for transformation, just as we want developers to harness existing theories in their designs and their validation.

Acknowledgements

We would like to thank our colleagues in the rCOS team (cf. <http://rcos.iist.unu.edu>) for their collaboration and discussions. We are in particular grateful to Anders P. Ravn for his suggestions and comments.

References

- [1] R. Back, L. Petre, and I. Paltor. Generalizing action systems to hybrid systems. In *Proc. of the 6th Intl. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*, pages 202–213. Springer, 2000.

- [2] G. Booch. *Object-oriented analysis and design with applications*. Addison-Wesley, 1994.
- [3] R. Burstall and J. Goguen. Putting theories together to make specifications. In R. Reddy, editor, *Proc. 5th Intl. Joint Conf. on Artificial Intelligence*, pages 1045–1058, Department of Computer Science, Carnegie-Mellon University, USA, 1977.
- [4] X. Chen, J. He, Z. Liu, and N. Zhan. A model of component-based programming. In F. Arbab and M. Sirjani, editors, *International Symposium on Fundamentals of Software Engineering (FSEN 2007)*, volume 4767 of *Lecture Notes in Computer Science*, pages 191–206. Springer, April 2007. UNU-IIST TR 350.
- [5] X. Chen, Z. Liu, and V. Mencl. Separation of concerns and consistent integration in requirements modelling. In J. van Leeuwen, G. F. Italiano, W. van der Hoek, C. Meinel, H. Sack, and F. Plasil, editors, *SOFSEM 2007: 33rd Conference on Current Trends in Theory and Practice of Computer Science*, volume 4362 of *Lecture Notes in Computer Science*, pages 819–831. Springer, January 2007.
- [6] Z. Chen, A. H. Hannousse, D. V. Hung, I. Knoll, X. Li, Y. Liu, Z. Liu, Q. Nan, J. C. Okika, A. P. Ravn, V. Stolz, L. Yang, and N. Zhan. Modelling with relational calculus of object and component systems—rCOS. In A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors, *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, chapter 3. Springer, 2008.
- [7] Z. Chen, Z. Liu, A. P. Ravn, V. Stolz, and N. Zhan. Refinement and verification in component-based model driven design. *Science of Computer Programming*, 2008. To be published.
- [8] Z. Chen, Z. Liu, and V. Stolz. The rCOS tool. In J. Fitzgerald, P. G. Larsen, and S. Sahara, editors, *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*, number CS-TR-1099 in Technical Report Series. Newcastle University, May 2008.
- [9] Z. Chen, C. Morisset, and V. Stolz. Specification and validation of behavioural protocols in the rCOS modeler. In *Proc. FSEN 2009*, 2009. To be published.
- [10] CWB. The concurrency workbench. <http://homepages.inf.ed.ac.uk/perdita/cwb/>.
- [11] E. Dijkstra. Notes on structured programming. In O.-J. Dahl, C. A. R. Hoare, and E. W. Dijkstra, editors, *Structured Programming*. Academic Press, 1972.
- [12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [13] P. Frederick and J. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.
- [14] P. Frederick and J. Brooks. The mythical man-month: after 20 years. *IEEE Software*, 12(5):57–60, 1995.
- [15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [16] J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of ACM*, 39(1):95–146, 1992.
- [17] J. He, X. Li, and Z. Liu. Component-based software engineering. In D. V. Hung and M. Wirsing, editors, *Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium*, volume 3722 of *Lecture Notes in Computer Science*, pages 70–95, Hanoi, Vietnam, October 2005. Springer. UNU-IIST TR 330.
- [18] J. He, Z. Liu, and X. Li. rCOS: A refinement calculus of object systems. *Theor. Comput. Sci.*, 365(1-2):109–142, 2006. UNU-IIST TR 322.

- [19] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [20] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
- [21] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
- [22] C. B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1990.
- [23] E. Y. Kang, Z. Liu, and N. Zhan. Component publications and compositions. Pro. 2nd International Symposium on Unifying Theories of Programming, Dublin, Ireland. To be published by Lecture Notes in Computer Science, 2008.
- [24] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice-Hall, 2nd edition, 2001.
- [25] D. Li, X. Li, J. Liu, and Z. Liu. Validation of requirements models by automatic prototyping. In *First IEEE Intl. workshop UML and Formal Methods*, J. Innovations in Systems and Software Engineering. Springer, 2008. To appear.
- [26] Z. Liu. Software development with UML. Technical Report 259, IIST, United Nations University, P.O. Box 3058, Macao, 2002.
- [27] Z. Liu, V. Mencl, A. P. Ravn, and L. Yang. Harnessing theories for tool support. In *Proc. of the Second Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 371–382. IEEE Computer Society, August 2006. Full version as UNU-IIST Technical Report 343.
- [28] Q. Long, Z. Qiu, and Z. Liu. Formal use of design patterns and refactoring. In T. Margaria and B. Steffen, editors, *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 323–338. Springer, 2008.
- [29] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.
- [30] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 1994.
- [31] Object Management Group. Unified Modeling Language: Superstructure, version 2.0, final adopted specification, 2005. <http://www.omg.org/uml/>, formal/05-07-04.
- [32] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, 44(10):43–50, 2001.
- [33] R. Paige, P. Brooke, and J. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Trans. Softw. Eng. Methodol.*, 16(3):11, 2007.
- [34] D. Parnas. On the criteria to be used to decompose systems into modules. *Communication of ACM*, 15:1053–1058, 1972.
- [35] L. Peter. *The Peter Pyramid*. New York, NY: William Morrow, 1986.
- [36] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, 1997.
- [37] A. Schneider. *The B-method*. Masson, 2001.
- [38] I. Sommerville. *Software Engineering (6th Edition)*. Addison-Wesley, 2001.
- [39] SRI. PVS specification and verification system. <http://pvs.csl.sri.com/>.
- [40] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
- [41] M. Wirsing, J.-P. Banâtre, M. Hölzl, and A. Rauschmayer, editors. *Software-intensive systems and new computing paradigms*, volume 5380 of *Lecture Notes in Computer Science*. Springer, 2008.

-
- [42] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.
- [43] L. Yang and V. Stolz. Integrating refinement into software development tools. In G. Pu and V. Stolz, editors, *1st Workshop on Harnessing Theories for Tool Support in Software*, volume 207 of *Electr. Notes in Theor. Comp. Sci.*, pages 69–88. Elsevier, April 2008.
- [44] L. Zhao, X. Liu, Z. Liu, and Z. Qiu. Graph transformations for object-oriented refinement. *Formal Aspects of Computing*, 21(1), 2009.