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

Zhiming Liu, Charles Morisset, and Volker Stolz

Intl. Institute for Software Technology (UNU-IIST),
United Nations University, Macao SAR, China
{lzm,morisset,vs}@iist.unu.edu
http://rcos.iist.unu.edu

Abstract. We present the roadmap of the development of the rCOS theory and its tool support 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 a discussion of the concepts, techniques and design decisions in the research of the theory and the development of the prototype tool. The concepts, techniques and decisions discussed here have been formalized and published. References to those publications are provided with explanations. Based on the initial experiences with a case study and the preliminary rCOS tool development, further development trajectory leading to further integration with transformation and analysis plug-ins is delineated.

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

1 Introduction

Complexity has long been recognized as an *essential property* of software, not an *accidental one* [12,13]. 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 [1]. The first attribute implies the challenges in the requirements capture and analysis and the problem of *changeability* of software to meet continuously changing requirements for additional functionality and features, the second one focus on the management of the development process and team, the third indicates the difficulties and creativity needed in making the right design decisions, 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

* Supported by the projects HighQSoftD and HTTS funded Macao S&TD Fund and the grants CNSF No. 60970031, NSFC No.90718014 and STCSM No.08510700300.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 62–80, 2010.

aircraft, cars, banks and supermarkets [5]. 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*.

For example, the CoCoME benchmark system [5] is a *trading system* for an enterprise of stores. It has components for *processing sales*, *management of different stores* and *management of enterprise*. These components are deployed on different computers in different places, and they interact among themselves and with external systems such as banks and product suppliers, through different middlewares (such as RMI and CORBA). The application software components have to be embedded with controllers of hardware devices such as product code scanners, printers, and credit card readers.

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 handling 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 in handling complexity by separation of concerns. 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.

2 Natural Path to CB-MDD

In software engineering, the complexity of a system development is mastered by *separation of concerns*. 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.

[1] Booch even calls this state of affairs "normal" [1].

2.1 Early Notions of Components and Models

In a waterfall process, the principles of *structured programming* [10] 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*, not only syntactically but also semantically . For this, each phase is required to produce *semantically 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 and trustworthy 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* [21,42,37], *event based* [18,30] and *property based* [28,29,22]. Development of *theorem proving* [39], *model checking* [20,36] and *simulation* [9] 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. It is important to note that development of tool support also requires the models formally described because tools are software and can only manipulate formally defined notations.

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*,
- 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.

For a safety critical application, it is further required that

- the models constructed are verifiable,
- model transformation are proven to preserve desirable properties of the models, such as functional correctness, and
- the model transformations generate conditions about the models, called *proof obligations*, that are verified by using verification techniques and tools.

This implies that what is critical to CB-MDD is a modeling approach with sound *theoretical foundation* and strong *technical and tool support*. The approach should integrate techniques and tools for *correct by construction* through model transformations and those for *analysis* and *verification*. The transformations support the engineering design and reduce the burden on automated verification. These form the theme of the rCOS method and in what follows we describe the key features of this modeling approach.

Multi-dimensional 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, 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 concerns[2] has to deal with the important issue of the *correctness* and *consistency* of *different views* [4,33]. 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 [3,6] 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 such as rCOS [17]. 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 are the key to the scalability of the method and 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,

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

we need to advance the ideas of "putting theories together" [2,15] and "unifying semantic theories of programming" [19] to produce, analyze and transform models with integrated tool support.

Object-orientation in CB-MDD. The rCOS method integrates object orientation as an important part. Among other reasons including reusability and maintenance, there are 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 [1]. *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*. This is found to be very useful for domain understanding, and requirements capture and analysis. 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 [23,6]. Secondly, the use of design patterns [14,23] makes the object-oriented design through model transformations more systematic and thus has much higher possibility for automation [17,27]. 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* [40,3,6]. 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 notion we find in formal methods that supports correct by design is *program refinement* [31,19]. However, the classical refinement techniques need to be generalized and unified to support the separation of concerns and allow models of different views

to be refined separately and hierarchically, such as data *functionality refinement*, *interaction refinement* and *object-oriented structure refinement* [17,3,44]. We need to scale up refinement rules and automate them via exploration of formal use [17,27] and automation of design patterns [14,23] for abstract models and refactoring rules [11] 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 of the models before and after the transformation. These verification tasks can be carried out by integrated verification tools, 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 take this view very important, as only with an engineering process 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. The rCOS tool, introduced in Sec. 4, intends to provide such an integrated tool suite and already allows one to run some model transformations and checking.

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 [17,3,6,44].

An essential concept in rCOS is that of *components* and rCOS provides a multi-dimensional approach to modeling a component. Along the *vertical dimension*, a component K has various models with different details, i.e. in different levels of abstraction, 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. On the horizontal dimension is the *hierarchical* and *dependency* relationships among *components*. It captures the composition of the component from sub-components with *connectors*, *coordinators* and *glue programs*. The third dimension separates the *different views* and characterize the different aspects of the component at the given level of abstraction, including the data and class structure, data functionality, interaction protocol, reactive behavior, etc.

3.1 Component Implementation and Component Refinement

At the code level, a component 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 methods that are called in the code of the component K, but not declared in the provided interface or defined as internal methods in the component.

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 [19]. 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 its says that if the program executes from a state where the *initial values* x satisfies $pre(x)$ the programs will terminates 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. The rCOS tool provides a textual language for specifying the methods, with a syntax mixing both designs (pre/post-conditions, non-deterministic choice, etc) and sequential code (conditional statement, loop, etc).

Semantics and refinement of components. With the definition of designs and the refinement calculus established in UTP, the semantics of a component K is defined as a function $\lambda C_{rIF} \cdot spc.K$. The type of the function is from the set of specification functions C of the required method methods in *K.rIF* to the specification functions of the required methods in *K.pIF*. For any specification function C (called a *contract* later) of the required interface *K.rIF*, $spc.K(C)$ is the function that gives each required method $n()$ a design $C(n())$ calculated in the calculus of UTP, and thus 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 [3], the *domain of reactive designs* $\mathcal{RD}$ is a sub-domain of the domain of designs $\mathcal{D}$ characterized by *lifting function* $\mathcal{H} : \mathcal{D} \to \mathcal{RD}$ such that $\mathcal{H}(p \vdash R) \mathrel{\widehat{=}} (true \vdash wait') \lhd wait \rhd (p \vdash R)$, stating that when

wait is *true* the execution stays in the *wait* state and proceeds according to the design $p \vdash R$ otherwise. 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 $\mathcal{H}(p \vdash R) \lhd g \rhd (true \vdash wait')$, where "$... \lhd g \rhd ...$" is the math infix operator operators for the programming construct **if** g **then** ... **else** ....

A component K_1 is a *refinement* of K, denoted by $K \sqsubseteq K_1$, 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 $spc.K(C)(m) \sqsubseteq spc.K_1(C)(m)$ holds in the refinement calculus of UTP. A work in progress is to automatically generate the proof obligations in Isabelle/HOL that a design is a refinement of another one.

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 [17,44]. 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. 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 black-box 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 $C = (I, \theta, \mathcal{F})$ 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 $\mathcal{F}$, and denoted by *C.spec*, specifying the reactive behavior by giving each method $m \in$ *C.IF.methods* a reactive design *C.spec*(m).

Note that the specification function *C.spec* combines the static (data) functionality view and the reactive dynamic behavior view [4]. A contract also has a third view, the *structure view*, that defines the data- and class structures. It is represented by a UML *class diagram*it, which defines the data types and classes of the objects of the component.

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 [3] by its *failures*, *failure.C*, and its *divergences*, *divergence.C* [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.

Correctness of components. It is now clear that $\lambda C_{rIF} \cdot$ *spc.K* calculates a contract of the provided interface of K for a given contract of C_r of its required interface. A component K *fulfills* or *implements* a contract C if there exists a contract such that $C \sqsubseteq$ *spc.K*(C_r). Clearly, *spc.K*(C_r) is the **s**trongest **p**rovided **c**ontract for C_r. We call a pair of contracts $C = (P, R)$ of *K.pIF* and *K.rIF* a *contract of component* K if $P \sqsubseteq$ *spc.K*(R), that is K fulfills the provided services P if the environment provides K with services R. We define the relation of *alternate refinement* between component contracts such that for publications $C_1 = (P_1, R_1)$ and $C_2 = (P_2, R_2)$, $C_1 \sqsubseteq C_2$ if $P_1 \sqsubseteq P_2$ but $R_1 \sqsupseteq R_2$, meaning that the refined component provides "better" services while requiring "weaker" services.

3.3 Publications

A contract of the interface of a component is good to be used by the designer of the component and for verification of the correctness of the design. However, it is rather operational for a user of the component. A user prefers a more declarative and more abstract specification in the form of a user manual. This is the notion of *publications* [16,25] for assembling components. Therefore, a publication is about the usage of the component.

In rCOS, a *publication* $P = (I, \theta, \mathcal{S}, \mathcal{T})$ specifies for an interface I, denoted by *P.IF*,

- an *initial condition* θ, denoted by *P.init*, that specifies the allowable initial states of the intended component,
- a specification function $\mathcal{S}$, and denoted by *P.spec*, specifying the *static data functionality* by giving each method $m \in$ *P.IF.methods* a design *C.spec*(m) (without guard), and

– a *protocol* $\mathcal{T}$, denoted by *P.prot*, that is a set of traces over the method names of *P.IF.methods*, specify the assumed *work flows* or *interaction protocol* in which the actors use services of the intended component.

We define that for a same interface I, a publication P_2 is a *refinement* of a publication P_1 of I if $P_2.init \Rightarrow P_1.init$, $P_1.prot \subseteq P_2.prot$ and for each interface methods m, $P_1.spec(m) \sqsubseteq P_2.spec(m)$.

In [25], a function $\mathcal{P}$ is defined to obtain a publication from a contract C, and a function $\mathcal{C}$ to obtain a contract from a publication; and the pair $\mathcal{C}, \mathcal{P}$ forms a *Galois Connection* between the domains of contracts and publications with respect to the refinement partial orders.

Component publications and their faithfulness. A *publication* of a component is a specification $U = (G, A)$, where G and A are publications of the provided interface *K.pIF* and required interface *K.rIF*, respectively. We define the relation of *alternate refinement* between component publications such that for publications $U_1 = (G_1, A_1)$ and $U_2 = (G_2, A_2)$, $U_1 \sqsubseteq U_2$ if $G_1 \sqsubseteq G_2$ but $A_1 \sqsupseteq A_2$.

We now extend the functions $\mathcal{P}$ and $\mathcal{C}$ to publications and contracts of components, i.e. to pairs of of publications and pairs of contracts, respectively. 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 for *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 [25], 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 design and verification of components while publications are used for substitutability and assembling.

Theorem of separation of concerns. The fact that $(\mathcal{P}, \mathcal{C})$ forms a *Galois connection* between the domain of contracts and the domain of publications makes the rCOS theory support the separation of concerns. It allows to preserve the faithfulness of a publication by refining the the data functionality and shrinking the protocol in of the provided publication while weakening the data function functionality and enlarge the protocol of the required required publication.

The notions of contracts and publications of an interface can be combined to a notion of *extended contract*[3] $C = (I, \theta, \mathcal{F}, \mathcal{T})$ which specifies the interface, the initial condition, reactive designs (behavior) and interaction protocol of an intended component. The protocol and the reactive behavior are therefore required to be consistent so that all traces in the protocol are allowed by the reactive behavior $\mathcal{F}$. Our experience with the CoCoME example [5] shows it is desirable for the design to specify a use case as an extended contract of the provided interface of a component (see Section 4 too).

[3] This is actually the notion of contract defined in [17].

The theorem of separation of concerns in [3] allows to refine the static data functionality and reactive behavior of a contract separately to preserve the consistency of an extended contract, and thus the faithfulness of a publication. It is interesting to point out that object-oriented refinement [17,44] 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.

3.4 Composition

The notion of *composition* is essential for a component-based design and must be defined for models of components at all levels of abstraction, and it should be consistently refactorable to composition of interfaces, static functionality, reactive behaviors and interaction protocols. As a *component-based architecture description langauge*, rCOS defines 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, and *disjoint parallel composition* of components. Internalization and plugging together can represent general components coordination. 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 *internalization* [25]. However, for the plugging composition is conditional. 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 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 methods must be aggregated into the remaining interface methods. In [3] a definition for internalization (called synchronization there) of a component by a *process* is given and the result proven to be a component. However, this is better used at the implementation level. In [25], internalization is directly defined for contracts and publications.

Composition of component implementations. The composition operators *renaming*, *restriction*, *plugging* and *disjoint parallel composition* are implemented as simple *connectors* following the semantics defined in [3,25]. Internalization is implemented by using *processes* (think of a scheduling process) that automatically calls the internalized methods 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 [3] 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 [6]. 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 [7]. 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. Therefore, the different views of a use case together form a model of an extended contract of a component whose provided interface provides the methods for the actors to call.

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 protyping from an analysis model [24] 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 [8]. 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 CoCoME case study [5] 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 threes 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 [23]. 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 o of class M into *knowing responsibilities* and *doing responsibilities* [23]. 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 $p \neq this$ 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 in the class M when needed:

$$\begin{array}{ll} find\_a\_b() \ \{return\ a.b\}, & find\_a\_m(\bar{p}) \ \{return\ a.m(\bar{p})\} \\ find\_m\_a(\bar{p}) \ \{return\ m(\bar{p}).a\}, & find\_m\_n(\bar{p}, \bar{q}) \ \{return\ m(\bar{p}).n(\bar{q})\} \end{array}$$

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 $p \neq this$ of type M, any attribute a and any expression e, we introduce the following rule:

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

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. [23] for informal presentation and [17] for an rCOS formalization) as well as the Creator Patterns, Structure Patterns and Behavior Patters (cf. [14] for informal description and [27] for the rCOS formalization) and refactoring rules (cf. [11] for informal discussion and [27,44] 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* [6]. 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 [6]. 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. [17,3,6]). 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 [6].

There is a long way to fulfill our vision on the design and the implementation of the tool set out in [26]. 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 story-telling 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. We also thank the anonymous reviewers for their comments.

References

1. Booch, G.: Object-oriented analysis and design with applications. Addison-Wesley, Reading (1994)
2. Burstall, R., Goguen, J.: Putting theories together to make specifications. In: Reddy, R. (ed.) Proc. 5th Intl. Joint Conf. on Artificial Intelligence, pp. 1045–1058. Department of Computer Science, Carnegie-Mellon University, USA (1977)

3. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 191–206. Springer, Heidelberg (2007)
4. Chen, X., Liu, Z., Mencl, V.: Separation of concerns and consistent integration in requirements modelling. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 819–831. Springer, Heidelberg (2007)
5. Chen, Z., Hannousse, A.H., Hung, D.V., Knoll, I., Li, X., Liu, Y., Liu, Z., Nan, Q., Okika, J.C., Ravn, A.P., Stolz, V., Yang, L., Zhan, N.: Modelling with relational calculus of object and component systems–rCOS. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 116–145. Springer, Heidelberg (2008)
6. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. Science of Computer Programming 74(4), 168–196 (2008); Special Issue on the Grand Challenge. UNU-IIST TR 388
7. Chen, Z., Liu, Z., Stolz, V.: The rCOS tool. In: Fitzgerald, J., Larsen, P.G., Sahara, S. (eds.) Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop, Newcastle University. CS-TR-1099 in Technical Report Series (May 2008)
8. Chen, Z., Morisset, C., Stolz, V.: Specification and validation of behavioural protocols in the rCOS modeler. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 387–401. Springer, Heidelberg (2010)
9. CWB. The concurrency workbench, `http://homepages.inf.ed.ac.uk/perdita/cwb/`
10. Dijkstra, E.: Notes on structured programming. In: Dahl, O.-J., Hoare, C.A.R., Dijkstra, E.W. (eds.) Structured Programming. Academic Press, London (1972)
11. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
12. Frederick, P., Brooks, J.: No silver bullet: essence and accidents of software engineering. Computer 20(4), 10–19 (1987)
13. Frederick, P., Brooks, J.: The mythical man-month: after 20 years. IEEE Software 12(5), 57–60 (1995)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
15. Goguen, J., Burstall, R.: Institutions: abstract model theory for specification and programming. Journal of ACM 39(1), 95–146 (1992)
16. Jifeng, H., Li, X., Liu, Z.: Component-based software engineering. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 70–95. Springer, Heidelberg (2005); UNU-IIST TR 330
17. He, J., Liu, Z., Li, X.: rCOS: A refinement calculus of object systems. Theor. Comput. Sci. 365(1-2), 109–142 (2006); UNU-IIST TR 322
18. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
19. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs (1998)
20. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
21. Jones, C.B.: Systematic Software Development using VDM. Prentice-Hall, Englewood Cliffs (1990)
22. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16(3), 872–923 (1994)

23. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice-Hall, Englewood Cliffs (2001)
24. Li, D., Li, X., Liu, J., Liu, Z.: Validation of requirements models by automatic prototyping. J. Innovations in Systems and Software Engineering 4(3), 241–248 (2008)
25. Liu, Z., Kang, E., Zhan, N.: Composition and refinement of components. In: Post event Proceedings of UTP 2008. LNCS. Springer, Heidelberg (to appear, 2009)
26. Liu, Z., Mencl, V., Ravn, A.P., Yang, L.: Harnessing theories for tool support. In: Proc. of the Second Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006), pp. 371–382. IEEE Computer Society, Los Alamitos (2006); Full version as UNU-IIST Technical Report 343
27. Long, Q., Qiu, Z., Liu, Z.: Formal use of design patterns and refactoring. In: Margaria, T., Steffen, B. (eds.) International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Communications in Computer and Information Science, vol. 17, pp. 323–338. Springer, Heidelberg (2008)
28. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems: specification. Springer, Heidelberg (1992)
29. Manna, Z., Pnueli, A.: The temporal logic of reactive systems: safety. Springer, Heidelberg (1992)
30. Milner, R.: Communication and concurrency. Prentice-Hall, Englewood Cliffs (1989)
31. Morgan, C.C.: Programming from Specifications. Prentice-Hall, Englewood Cliffs (1994)
32. Ossher, H., Tarr, P.: Using multidimensional separation of concerns to (re)shape evolving software. Commun. ACM 44(10), 43–50 (2001)
33. Paige, R., Brooke, P., Ostroff, J.: Metamodel-based model conformance and multiview consistency checking. ACM Trans.Softw.Eng.Methodol. 16(3), 11 (2007)
34. Parnas, D.: On the criteria to be used to decompose systems into modules. Communication of ACM 15, 1053–1058 (1972)
35. Peter, L.: The Peter Pyramid. William Morrow, New York (1986)
36. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1997)
37. Schneider, A.: The B-method. Masson (2001)
38. Sommerville, I.: Software Engineering, 6th edn. Addison-Wesley, Reading (2001)
39. SRI. PVS specification and verification system, `http://pvs.csl.sri.com/`
40. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Reading (1997)
41. Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.): Software-Intensive Systems and New Computing Paradigms. LNCS, vol. 5380. Springer, Heidelberg (2008)
42. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Englewood Cliffs (1996)
43. Yang, L., Stolz, V.: Integrating refinement into software development tools. In: Pu, G., Stolz, V. (eds.) 1st Workshop on Harnessing Theories for Tool Support in Software. Electr. Notes in Theor. Comp. Sci., vol. 207, pp. 69–88. Elsevier, Amsterdam (2008); UNU-IIST TR 385
44. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects of Computing 21(1-2), 103–131 (2009)