The rCOS tool⋆

Zhenbang Chen1,2, Zhiming Liu1, and Volker Stolz1

1 International Institute for Software Technology
United Nations University, Macao SAR, China
2 National Laboratory for Parallel and Distributed Processing, China

Abstract. The goal of the rCOS tool is to harness state of the art techniques from use case- and model driven-development of component-based systems on top of UML. It facilitates both the development process and the persistence of formal verification artefacts in the model: use cases are specified in multiple views, using Class, Sequence and State Diagrams to capture their behaviour. Functionality is specified in rCOS pre-/post-conditions based on UTP. Provably correct refinement steps transform the Requirements Model into a component-based Design Model.

Consistency of the different views is ensured, for example by checking that the State Diagram accepts the protocol specified in the Sequence Diagram. For component composition, protocol compatibility is verified. To that end, either third party tools like the FDR2 model checker are invoked, or annotations to code skeletons for additional tools like the Java Modeling Language (JML) are generated. The tool is implemented on top of the Eclipse platform using a UML profile, ensuring compatibility with other UML-based software engineering tools.

1 Introduction

The software development process is no longer an activity a single programmer can carry out from beginning to the end. For non-trivial systems, a Use Case-driven approach requires a variety of specification and implementation tasks: based on an informal requirements document, firstly a domain model of the system has to be developed. It is traditionally captured in a Class Diagram for object-oriented languages.

Additionally, the behaviour in terms of possible interactions that the system offers to the environment is specified through diagrams like Sequence Diagrams. Internally, the behaviour may be specified in State Machine Diagrams, detailing how the system changes its state when its provided services (usually methods) are invoked.

Only after the specification is complete, design of the actual methods is started. Here, ideally also a mathematical or at least platform-independent specification is created first, that is then refined into the platform-specific code. Preferably, some mechanisms like testing or even verification techniques ensure that the implementation fulfills the specifications.

A model driven development process [21] should make it easy to evolve and integrate changes into the system based on modification of the diagrams: a change in the specification should be propagated automatically downstream until code generation. Repeated testing and/or verification becomes necessary.

On top of the object-oriented modeling, another layer of abstraction can be added: components are supposed to facilitate building an application out of prefabricated parts that can be used only knowing their specification [25]. Components

⋆ This work is supported by the projects HighQSoftD and HTTS funded by Macao Science and Technology Development Fund; the 973 Project of China 2005CB321802; the 863 Project of China 2006AA01Z165; and the NSFC Projects 60673114, and 90612009.
can be plugged together according to their interfaces. Existing components can be refactored by their designer into smaller components, just like classes.

As component interfaces can contain exceedingly complex contracts that describe their behaviour and interactions, checking the validity of component composition is by no means trivial. The most prominent example of this concept is probably Meyer’s “Design-by-contract” [18], which started out specific to the Eiffel programming language, but is now also used in other languages such as Java. While the necessity of contracts has been widely realised, there is no unified modeling and verification approach, yet.

The rCOS requirements- and design modeling tool aims to offer an integrated environment on top of the Eclipse platform [28] for model driven development of component-based systems from specification and refinement over to verification and code generation. The theoretic foundation is the “Refinement of Component and Object Systems” (rCOS) that extends Hoare and He’s Unified Theories of Programming [16, 12, 10].

Various UML [23] diagrams contain the specification of an rCOS requirements model which is refined into a design model. Static and dynamic consistency of those models is assured by formal methods. For the component model, component composition with contracts based on the rCOS calculus can be verified. The main process of our tool is shown in Fig. 1.

![Diagram](image-url)

Fig. 1. The main process of rCOS tool

In the rCOS development process [5], the “component” plays the center role. Different modeling elements of components (Interfaces, Ports, Delegation) can be used to specify use cases at the requirements modeling stage. After requirements modeling, the object-oriented design is to be done by applying refinement rules. The logical component-based architecture design is carried out after the object-oriented design stage. The designer composes or decomposes the components with respect to the nature of the classes and objects in the previous design stage, and the compatibility of the compositions must be verified. Finally, the code generation and detailed design generates platform dependent code skeletons and may require implementation of the significant algorithms in the specification. Our tool currently supports the requirement modeling stage, and we are working on the implementation of the refinement rules and verification backends.
Related Work. Olderog et al. present CSP-OZ, a formal method combining CSP
with the specification Object-Z, with UML modeling and Java implementations [19].
They use a UML profile to annotate the model with additional data. Model properties
then be verified on the CSP, and their notion of contracts of orderings between method
invocations through JML and CSP_jassda can be enforced on Java
programs at runtime. Their tool Syspect is also built on the Eclipse Rich Client
Platform.

The VDM (Vienna Development Method) tool chain [7] and its open-source
spin-off “Overture” [29] combine object-oriented modeling with the analytic power
of formal methods. Modeling is supported in UML, and code generation is possible
to Java and C++.

TOPCASED [24] is an Eclipse-based open-source toolkit to develop different
models, e.g. in UML and make them accessible to analysis and verification by tools.
It serves as a “Model Bus” between the different front- and backends for verification
and code generation.

Executable UML [17] introduces a UML profile that gives a suitable semantics
for direct execution to a subset of UML. As such, it focuses on execution and not
formal verification. Use cases and State Diagrams are used, procedures are specified
in an action language.

Outline. The paper will be organized as follows: Section 2 explains Requirements
Modeling in detail and the verification issues arising during the development process;
in Section 3, the technical aspects and limitations of the tool are discussed; finally,
the conclusion and the future work are given in Section 4.

2 Requirements Modeling

In the requirements modeling stage of Software Development [14], the problem
domain of the use case is analysed and the artefacts of this stage are created.

In the component-based model-driven development process, it lets developers
design the system at a high level of abstraction from the beginning by using and
creating models or specifications of components. rCOS has a strong notion of compon-
ents that exceed what is usually offered in current programming languages like e.g.
Java: a component is an aggregation of classes together with a contract attached to
each component interface, and the contract contains an rCOS protocol
specification [1].

Firstly, each use case is taken as a component, and the functionalities of a use
case are modeled as methods in the provided interface of a component. The com-
ponent aggregates the relevant classes and their associations taking part in the use
case, and a conceptual class diagram is derived from the problem description. This
may require some interpretation of the description by a domain expert. We borrow
the term “conceptual” class diagram from Larman [14] to indicate that at this stage,
we do not assign visibility to the attributes and assume that they are all public.
Also, there are initially no methods except for the controller class implementing the
provided interface. The others will be introduced through refinement steps in later
stages.

Secondly, the sequence diagram illustrates the interaction of the user with the
system, which will have to conform to the protocol in the component contract. We
allow only a limited use of the UML sequence diagram (collaboration) facility: there
is only one actor (the user) and one process (the system). Messages only flow from
the user to the system and represent invocations of methods in the component inter-
face. Any information that is required for input or output, e.g. presented to the user
on a display is implicit and will be modeled in the next stage. We allow the usual
control structures such as iteration and conditional branches in the sequence diagram. These have controlling expressions in the form of boolean queries or counters. The introduced methods must exist in the component interface and the conceptual class diagram, and can be created on the fly when adding a new message (this is a common feature of other graphical modeling tools such as MagicDraw [20] as well).

While the sequence diagram describes the possible interactions with the system the user can have, the state diagram describes how the system internally changes state during execution. Edges in the UML state machine diagram are labelled with an expression of the form $g \& m(x_1, \ldots, x_n)$, indicating that this edge may be triggered by an invocation of method $m$ if the guarding expression $g$ is evaluated to be true. The usual notions of well-formedness for those expressions, e.g. visibility of attributes, are applied. We allow nondeterminism by having multiple outgoing edges from the same state, each labelled with the same method and potentially overlapping guards for specification purposes, but point it out to the user as potentially undesired in the light of a future implementation of the protocol. There is a single initial state; we currently do not allow edges without a method. Missing guards are assumed to be true. State labels do not carry any semantics and are just informative; they may be used e.g. to generate labels in subsequent stages. Choice nodes with a boolean expression allow if-then-else constructs, without such an expression we treat all alternatives in a non-deterministic internal choice.

Lastly, the logic of actual actions to execute when a method is called is given in the Functionality Specification. Each method in the control class has its pre-/post-conditions specified in rCOS with the appropriate static checking of syntax, types, etc. The guard of each method can be calculated from the guarding expressions in the state diagram by the method in [2].

Naturally, there is a close relation between the trace languages induced by the sequence diagram and the state diagram. The state diagram must at least accept the runs of the sequence diagram. Conversely, the problem description should specify if the state diagram is allowed to offer more behaviour than the sequence diagram. This may also be an issue if synthesis of multiple, related use cases into a single, more complex diagram is considered—at the moment, rCOS does not have this provision.

**Refinement**

A use case in the requirement model can be refined to incorporate sub-use cases. As the result of this, the corresponding component must also be refined to contain sub-components. Invariants can be specified for the parent component to constrain its sub-components, e.g. for an office system that both front- and back-office are actually connected to the same database [5]. Fig. 2 is a screenshot of a refined component view.

Also, the Requirements Model will be refined to a Design Model in various aspects: through the Expert Pattern [6], methods in the sequence diagram are delegated to other classes and their methods. Aggregating multiple classes into a single rCOS component is also a refinement step.

To apply the Expert Pattern, a new message is derived from a selected message in the sequence diagram. The new message is inserted immediately after the reception of the selected message, going from the original receiver to the new receiver (represented as a lifeline different from the original sender) in the diagram. The new receiver can either be backed by a (newly introduced) class, or it can refer to an already existing component with the corresponding method in its interface. The new message does not need to have the same name or the same arguments as the original message.
The functionality specifications are refined to platform-independent code through further transformation steps, except where an entire algorithm has to be developed. This makes use of patterns e.g. to resolve existential or universal quantification in an rCOS specification.

The overall result of this stage is a component model through stepwise refinement that provably refines the initial Requirements Model. Many of the pre- and post-conditions can then almost trivially be refined into executable (platform-specific) code. Code generation can make use of platform-specific patterns e.g. for collection classes handling sets. For a concrete deployment of a system, code generation also has to choose either local or remote method invocation for communication.

Verification

The main advantage of the rCOS methodology is that we can automatically assure consistency of the multi-view specifications [15], for example by checking trace equivalence or deadlock freedom of the diagrams. We generate appropriate CSP [11] specifications (see [3]) for the FDR2 model checker [8]).

Another aspect of automated verification is component composition, that likewise we handle through CSP specifications. We can check the provided and required contracts by combining the components through CSP’s alphabetized parallel operation and hiding, which again yields a new component with a new contract [9]. Again we can check for trace equivalence or deadlock freedom of such a composition in FDR2.

In a related paper, the practicability of generating a PROMELA specification for the Spin model checker [32] from rCOS has been investigated. The semantics of the rCOS (PROMELA) specification is derived from executing the main method, which also initializes e.g. a concurrent system. The model is executed by the model checker for a bounded number of objects and invariants are checked. We intend to integrate this with the presented tool.

Additionally, the preservation of any application property during modeling after each refinement step can be guaranteed by different methods. The correctness of the
refinement during the object oriented design stage is guaranteed by the refinement rules that are already proved [10]. The correctness of platform-dependent code generation can be verified by using the techniques of data refinement [13], and static checking and runtime verification can be used at the detailed design stage to assure the correctness of implementation. We have tried JML by manual translation [6], and plan to integrate the JML tool suite with the current tool.

We can thus assure a provably correct model driven development and verification process of a component-based system from design to code generation.

3 Technical Notes

We use UML and Model Driven Development and Architecture (MDD/MDA) techniques [21] ourselves to develop our tool. While our tool is not yet complete and might not be amenable to solve a task which is not use case-driven (like our application itself), we also apply the discussed techniques to ensure correctness (or rather increase trust in a correct design) and validate our design against Software Engineering best-practices. In the following, we discuss some of the techniques employed and their limitations.

For the modelling part, we use UML 2 models and a UML profile to tie together the necessary information, e.g. by assigning which diagrams belong to a use case. The diagrams are bundled in a Package and tagged with “boolean” stereotypes from the profile. UML 2 already specifies some stereotypes like those for pre- and post-conditions on methods or guards of state machine diagram transitions. Occasionally, new tags are introduced (see the relations of the top-level RCOSModel stereotype).

The profile is documented in Fig. 3. This diagram mixes UML metamodel classes (Port, Component, . . . , dark) with the classes obtained by stereotype application (light). Apart from the collection of diagrams near the top of the diagram, the other important area are the ContractInterfaces attached to components.

Documentation of the profile facilities also turns out to be problematic: due to the reliance on UML packages to group elements, the Profile Diagram in Fig. 4
itself does not contain enough information to serve as documentation. We invite the reader to compare the actual UML profile in Fig. 4 with the handwritten and more detailed illustration in Fig. 3—in the former, only the stereotype-information is preserved, since all modelling elements already present in UML like Use Cases, Classes, etc. may not be repeatedly presented in the UML profile. As virtually every element can go into a UML Package, for example the explicit link in the first diagram from a Component Model (a stereotype for a Package) to a Service Component (subtype of Component) is not represented. It is very hard to discover the actual profile applications necessary from the rCOS profile when not using our own graphical tool, which only allows the user to describe conforming models in the first place. Conversely, even in our tool we have to enforce more restricted diagrams to be able to assign a meaning in rCOS to them (e.g. not every UML component diagram with rCOS stereotypes is a well-formed rCOS component diagram).

Stereotypes are also using in a component diagram to assign rCOS contracts to interface. Here, we encounter that it is occasionally hard to make rCOS and UML converge: ideally, we would like to attach an rCOS contract to the interface. In most UML modeling tool, you would usually expect to see two components connecting to a single interface. However, due to the notion of a required and a provided interface (the required interface may be a subset of the provided interface), in the graphical notation two components cannot share the same UML interface in the diagram, and thus we need to introduce intermediate interfaces that the contracts can be attached to. Otherwise, it would not be possible to find out which component provides which contract (see Fig. 2).

Although this is in itself not much of a problem, it may seem counter-intuitive to the user. Another issue arises when we need to record data in the model that is not readily captured in UML terms: if the user wants to associate a handwritten contract with a component, e.g. given as a regular expression, it needs to be stored in a textual representation (as string) and parsed/serialized as necessary. We expect this to become necessary more frequently as we introduce the more advanced features from rCOS like refinement/transformation rules.

This is similar to combining the Java programming language with UML: UML captures the static structure (in the form of a class diagram) easily, but generally cannot give an executable specification of the method bodies.

Fig. 4. rCOS profile diagram
Intermediate representations within our application are occasionally based on the Eclipse Modeling Framework (EMF), allowing us to document or even implement parts of the application in the Object Constraint Language (OCL) [30].

The graphical interface is built on top of TOPCASED [24] (see Related Work). We also try to maintain compatibility with the Eclipse UML 2 Modeling Tools and MagicDraw.

4 Summary and Outlook

We have presented an overview over the rCOS modelling tool for component-based model-driven development. Our tool harnesses the power of formal techniques to achieve a (potentially provably correct) use case-driven design process.

Based on UML and UML profiles, an rCOS requirements model is specified diagrammatically as a use case with State-, Sequence-, and Class Diagrams. The model also contains additional information such as rCOS contracts. Consistency of the design and component composition is assured by third party tools like the FDR2 model checker. Code generation allows for an easy creation of model- and behaviour skeletons. The generated code will be augmented by JML annotations and runtime checks.

As we come to grips with the already existing techniques like OCL, we hope to use them even in the generated code. In a second step, we hope to be able to specify our own tool with the tool as a proof of concept—although we expect that the crucial bits would be within the scope of significant algorithm design, i.e. maintaining and processing data structures. This usually requires high-level tools like again OCL, but cannot be automated.

For an industrial-strength development process, based on the existing commercial model driven development tool MasterCraft [26] we discuss different roles and responsibilities in the process. In practice, development tasks are already split between managers, designers and programmers, and we propose to implement additional roles taking care of verification and analysis [15].

rCOS specifies a large collection of refinement rules like the Expert Pattern or Data Encapsulation that we previously applied manually in a case study [3, 4]. We intend to implement these rules in the tool.

This is closely related to the field of Model Transformation: We intend to use the relational model transformation language QVT (for “Query/View/Transformation”, standardized by OMG [22, 27]) to implement the refinement rules [31].

An ongoing topic of research is how those rules affect properties like class invariants, that might be specified for an rCOS program.

The usability of the tool in combination with the rCOS design methodology will be validated with students in a summer school. UNU-IIST and the rCOS project hope to increase the awareness for Formal Methods in teaching and practice through dissemination of the tool. The tool is freely available as a bundle of Eclipse plugins from http://rcos.iist.unu.edu.

References


