Specification and Validation of Behavioural Protocols in the rCOS Modeler

Zhenbang Chen[1,2], Charles Morisset[1], and Volker Stolz[1]

[1] United Nations University
Institute for Software Technology
P.O.Box 3058, Macau SAR
{zbchen, cm, vs}@iist.unu.edu
[2] National Laboratory for Parallel and Distributed Processing
Changsha, China

Abstract. The rCOS modeler implements the requirements modelling phase of a model driven component-based software engineering process. Components are specified in rCOS, a relational calculus for Refinement of Component and Object Systems. As an aid to the software engineer, the modeler helps to separate the different concerns by creating different artifacts in the UML model: use cases define a scenario through a sequence diagram, and methods are given as guarded designs in rCOS. rCOS interface contracts are specified through state machines with modelling variables. Messages and transitions in the diagrams are labelled with method invocations.
The modeler checks the consistency of those artifacts through the process algebra CSP and the model checker FDR2: a scenario must follow a contract, and an implementation must not deadlock when following the contract. We illustrate the translation and validation with a case study.

1 Introduction

Software engineering is becoming more complex due to the increasing size and complexity of software products. *Separation of concerns* is an effective means to tackle modelling of complex system. The quality of a system can be improved by applying formal techniques in different development stages.

In our previous work, we introduced the notion of an *integrated specification* that derives a specification of a component-based system from a UML-like model for a use case [2]. A use case defines a syntactic *interface* (the provided methods) and controller class implementing this interface plus its referenced data structures, a system *sequence diagram* involving only a single actor and the component that describes the external behaviour, and a *state machine* describing the internal behaviour of the component. In the sequence diagram and the state machine, guarded transitions are labelled with methods from the interface.

Apart from the usual notion of well-definedness (also called *static consistency*) for an object-oriented design, we require the *dynamic consistency* of the specification: the state machine must accept all interaction sequences described by the system sequence diagram, and any implementation of the interface must not deadlock if invoked according to the *protocol* given through the state machine.

We have implemented the requirements modelling stage of a use case-driven model-based component development process following the rCOS methodology in the rCOS modeler. An rCOS model is a UML model extended through the rCOS UML profile [5]. The tool supports static checking of the dynamic consistency of the model through semi-automated translation into the process algebra CSP and the model checker FDR2 [19, 6].

We define an automated abstraction into 'flat' rCOS that only uses primitive types. The abstraction is further parametrized according to criteria specified by the user, e.g. to hide method- (and thus message-) arguments/return parameters. This flat representation is then translated into CSP, which we use to check that the generated rCOS design follows the protocol by checking the deadlock freedom of parallel composition of the generated CSP processes.

The paper is organized as follows: Section 2 presents our approach of separation of concerns including the underlying theory, the used modelling artifacts, and a brief explanation of integrated specification and checking; we explain the dynamic consistency checking in Section 3, which contains the translation from the rCOS model to CSP and the abstraction method used in the translation; finally, Section 4 concludes.

Related work Olderog *et al.* present CSP-OZ, a formal method combining CSP with the specification Object-Z, with UML modelling and Java implementations [12]. They use a UML profile to annotate the model with additional data. Model properties can then be verified on the CSP, and their notion of contracts of orderings between method invocations through JML and $\text{CSP}_{\text{jassda}}$ can be enforced on Java programs at runtime. Their tool `Syspect` is also built on the Eclipse Rich Client Platform.

Executable UML [11] 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.

The practicability of generating a PROMELA specification for the Spin model checker from rCOS has been investigated in [22]. The semantics of the rCOS (PROMELA) specification is derived from executing the `main` method. The model is executed by the model checker for a bounded number of objects and invariants are checked.

Snook and Butler [20] use B as the underlying theory during the modeling and design process using UML. The class diagrams and state diagrams in UML can be translated to a B description, including the function specifications and guards for operations. They also use a state variable in B to represent transitions during the translation of a state diagram. The refinement notion in B supports refinement between different UML models in the development stages. Ng and Butler discuss a similar translation of UML state machines to CSP in [14, 13].

Our concern in this paper is checking the consistency of multi-view specifications. The translation to CSP also extends to the verification of component composition in the rCOS theory (see [1]).

2 Separation of Concerns

The rCOS language is based on UTP, Unifying Theories of Programming [8], and is object-oriented. We give here a brief description of its main features. We refer to [4] for further details.

Method A method $m \in \mathit{Meth}$ is a tuple $m = (\mathit{Name}, \mathit{in}, \mathit{out}, g, d)$ where Name is the name of the method, in (resp. out) is a set of input parameters together with their type, $g \in \mathbb{G}$ is the guard and $d \in \mathbb{D}$ is the design. A guard is a boolean expression, which does not contain any primed variable from the post-state, and cannot refer to parameters of the method. A design could be

- a pre/post-condition pair $[p \vdash R]$, where p and R are predicates over the observables,
- a conditional statement $d_1 \lhd e \rhd d_2$, where d_1 and d_2 are designs and e is a boolean expression,
- a sequence $d_1; d_2$, where d_1 and d_2 are designs,
- a loop $\star(e, d_1)$, where d_1 is a design and e is a boolean expression or
- an atomic command, such as an assignment, a variable declaration, a method call, SKIP or CHAOS.

Interfaces An *interface* $I \in \mathbb{I}$ is a tuple $I = (\mathit{FDec}, \mathit{MDec})$ where FDec is the *fields declaration section* and MDec the *method declaration section.* Each signature is of the form $(m, \mathit{in}, \mathit{out})$, where in are input parameters and out are the output parameters.

Class A class $c \in \mathbb{C}$ is a tuple $c = (\mathit{FDec}, \mathit{MSpec})$ where FDec is a set of fields and $\mathit{MSpec} \subseteq \mathit{Meth}$ a set of methods. We assume that we can project onto the public and private attributes (or methods) of interfaces (and classes respectively) through $I.\mathit{FDec}_{pub}$ and $I.\mathit{FDec}_{priv}$.

Contracts of Interfaces A *contract* $\mathit{Ctr} = (I, \mathit{Init}, \mathit{MSpec}, \mathit{Prot})$ specifies

- the allowable initial states by the initial condition Init, a predicate over the attributes in I.
- the synchronization condition g on each declared method and the functionality of the method by the specification function $\mathit{MSpec} : I.\mathit{MDec} \rightarrow (\mathbb{G} \times \mathbb{D})$ that assigns each method defined in I to a guarded design $g\&D$.
- Prot is called the *protocol* and is a set of sequences of call events; each is of the form $?op_1(x_1), \ldots, ?op_k(x_k)$. A protocol can be specified through many different means, here we will consider protocols generated from sequence diagrams and state machines.

UML-based Requirement Specification

The models are defined in the Unified Modeling language (UML), and we apply Model Driven Development and Architecture (MDD/MDA) techniques [16].

For the modelling part, we use UML models and a UML profile to tie together the necessary information, e.g. by assigning which specification belongs to a use case. The specifications are bundled in packages and tagged with stereotypes from the profile to mark them as belonging to the rCOS modeling domain. The profile is documented in [5].

The advantage of using UML is three-fold: firstly, we can provide the familiar modelling notations for the system developer, yet augmented with a rigorous underlying mathematical semantics; secondly, we can reuse the UML meta-model by using a profile to get an rCOS meta-model, because rCOS intentionally overlaps with UML; lastly, there are numerous tools and methodologies for UML, and UML models are the de facto standard models that they create or exchange, so we can harness their powers to enhance the rCOS tool, e.g. we support importing from the UML tool MagicDraw [15], and also re-used an existing graphical UML modeler, saving us development effort.

2.1 Example

We use one use case (called ProcessSale) of our recent case study [3] as the example, which is a trading system based on Larman's textbook [9], originally called the *Point of Sale* (POS) system. The trading system records sales, handles both cash payments and card payments as well as inventory management. The system includes hardware components such as computers, bar code scanners, card readers, printers, and software to run the system. The *normal courses* of interactions in the *ProcessSale* use case are informally described:

1. The *cashier* sets *checkout mode* to express check out or normal check out.
2. When a *customer* comes to the *checkout* with their *items* to purchase, the *cashier* indicates the system to handle a new *sale*.
3. The *cashier* enters each item, either by typing or scanning in the *bar code*; if there is more than one of the same item, the cashier can enter the *quantity.* The system records each item and its quantity and calculates the subtotal. When the cash desk is operating in *express mode*, only a predefined maximum number of items can be entered.
4. When there are no more items, the *cashier* indicates to the system *end of entry.* The *total* of the sale is calculated.
5. The customer can pay by cash or credit card. If by cash, the amount received is entered. In express mode, only cash payment is allowed. After payment, the inventory of the store is updated and the completed sale is logged.

There are *exceptional* or *alternative courses* of interactions, e.g., the entered bar code is not known in the system. At the requirements level, we capture these exceptional conditions as preconditions.

2.2 Integrated Specification

In the *requirements modelling* stage of software development, each use case is modelled as a component, with the specification of the contract of its provided interface containing the interaction protocol, the reactive behaviour, and the functionalities the methods provided in the interface. The main advantage of the rCOS methodology is that we can assure consistency of the multi-view specifications [10], for example by checking trace equivalence or deadlock freedom of the diagrams. We generate appropriate CSP specifications [3] for the FDR2 model checker [7, 6]. While looking at state machines for verification, combining

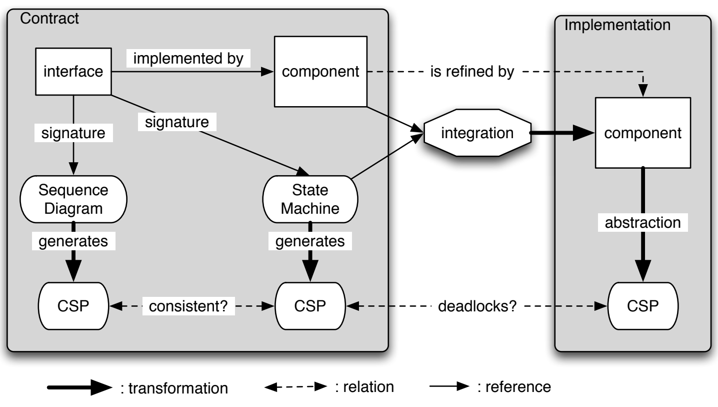

Fig. 1. Overview of specification and checking

UML and CSP is certainly not new, cf. [14], but we hope to make state diagrams more prominent in a formal development process, and we also consider guarded designs. The overview of the integrated specification and checking is shown in Fig. 1: the interface serves as the signature of a contract that contains a sequence diagram for the scenario and the state machine for the protocol. A component can implement a contract, and a component can be refined. We later discuss the different operations like generating CSP from the different artifacts, or integrating the contract with a functionality specification (a class containing the business logic) into a guarded design that implements the contract. Together with appropriate helper functions for abstractions, we can then turn an implementation into a ‘flat’ rCOS program. We can then use again CSP to verify that the implementation actually follows a contract. This is interesting for implementations coming from third parties that claim to respect a particular contract.

The component in the contract box aggregates the relevant data like objects, classes, and their associations taking part in the use case. The data types and classes are modelled as a class diagram that is derived from the problem description. We borrow the term “conceptual” class diagram from Larman [9] 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, and each method of the controller class can be specified with a guard and the function specification. The function specification of the *enterItem* method in the control class *CashDesk* for the *CashDeskIF* contract of the *ProcessSale* use case is as follows.

```
public enterItem(Barcode code, int qty ; ) {
  VAR LineItem item ;
  [ pre  : store.catalog.find(c) != null  ,
    post : line' = LineItem.new(c, q) ;
           line.subtotal' = store.catalog.find(c).price * q;
           sale.lines.add(line)]  ;
  [ ⊢  itemCounter' = itemCounter + 1 ]
}
```

In rCOS, the notation $[p \vdash R_1; R_2]$ stands for $[p \vdash R_1]; [\mathit{true} \vdash R_2]$ and that for each post-condition, there is an implicit statement leaving all the variables non concerned by the post-condition unchanged.

For different concerns, the *sequence* and *state diagrams* illustrate 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 interface. 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.

While the sequence diagram describes the possible interactions with the system the user can have, the *state machine* describes the *contract* of the provided interface. Edges in the UML state machine are labelled with an operation of the interface, which may have the form g & $op(\bar{x}; \bar{y})$, indicating that this edge may be triggered by an invocation of method op iff the guarding expression g is evaluated to be true. 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. Choice nodes with a boolean expression allow if-then-else constructs, or non-deterministic internal choice (not present in example). The protocol of the *CashDeskIF* contract is modelled by the sequence diagram in Fig. 2 and the dynamic flow of control by the state diagram in Fig. 3.

Naturally, there is a close relation between the *trace languages* over the method calls induced by the sequence diagram and the state diagram: the state machine must at least accept the runs of the sequence diagram. Conversely, the problem description should specify if the state machine is allowed to offer *more behaviour* than the sequence diagram.

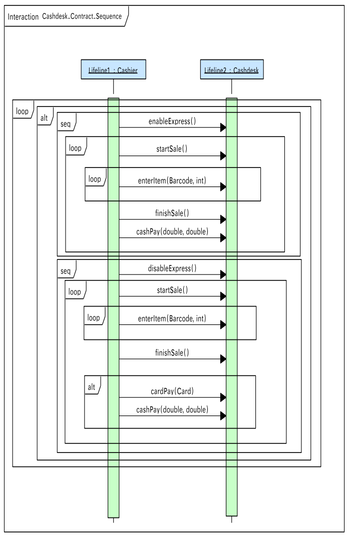

Fig. 2. Sequence Diagram

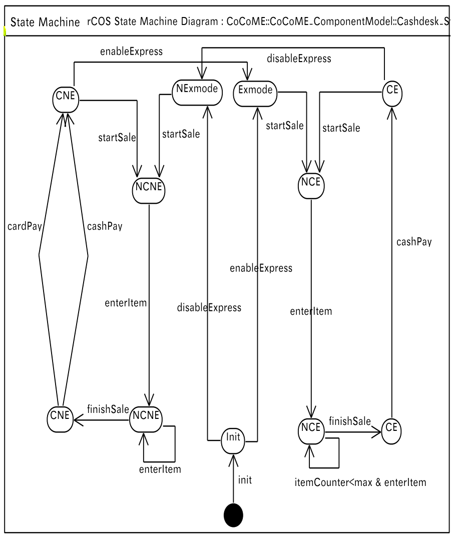

Fig. 3. State Diagram

3 Verification

For the translation of UML state machines into CSP, we limit ourselves to a subset of the features available in UML. Some of the limitations are arbitrary and based on the practical diagrams that we use in the case study. Some obvious extensions and shorthands are left open as future work.

We only allow a single initial state and plain states, connected by transitions. State labels do not carry any semantics and are just informative; they may be used e.g. to generate labels in subsequent stages.

Given a contract *Ctr*, transitions between states are labelled with an operation $op(\bar{x}; \bar{y}) \in I.MDec$ from the associated interface I of the contract that have a 'flat' guarded design $g_{Ctr_{op}}$ & $D_{Ctr_{op}}$. In an integrated specification, the operation *op* is mapped to a guarded design $g_{C_{op}}$ & $D_{C_{op}}$ in a class C. The guarding expression $g_{Ctr_{op}}$ may only refer to private attributes of the interface (modelling variables), and g_{op} only to members of the class. Missing guards are assumed to be *true*. We do not use UML's effects to designate an activity that occurs when a transition fires, but will instead use the method specification from the contract.

For the translation to CSP, the rCOS guarded designs must be mapped into CSP. As this is generally hard to automate because some abstraction needs to be applied, we proposed that this step is done by a verification engineer [10].

We simplify and automate things as much as possible in the tool: for example, constructs over integers (or sets thereof) can usually be trivially abstracted into CSP. We expect that modelers use mostly primitive types in guards of state machines and sequence diagrams. Also the guarded design derived from an integrated specification uses integers to identify the current and successor state.

Objects and object references of course pose a problem, since CSP has no concept of object orientation. Thus objects and any use of navigation in complex expressions must be broken down. Since we can handle 'flat' rCOS, that only uses primitive types, more easily as described above, we solve the problem by allowing the verification engineer to define an *abstraction function* $\alpha : rCOS \rightarrow rCOS$. α should preserve the refinement relation, that is formally in rCOS, $\forall m \in rCOS : m \sqsubseteq \alpha(m)$. That way, we translate an object oriented rCOS specification into CSP by going through an intermediate step where the verification engineer applies the abstraction function (in this case from object-oriented to 'flat' rCOS), from which we then generate the CSP. We intend to provide a library of abstractions, for example, only considering object identifiers (again in the form of integer values) for objects, and for common types like sets and bags. For the latter, there can be different granularity, like only considering if such a structure is empty or if it *may be* full in a three-valued abstraction. Another simple abstraction is hiding variables. Since the result has to be another valid rCOS specification, all expressions referring to the now hidden variables have to be properly abstracted as well. Currently it is the responsibility of the user to consider the implications of his chosen abstraction with regard to soundness and completeness of any property of the abstracted model, and the impact on the size of the state space of the model.

Let $\mathbb{C}\textsc{sp}$ denote a syntactically and statically valid CSP specification consisting of a set of process definitions. We consider now contracts in $\mathbb{C}tr_{SM}$, that are contracts for which the protocol is defined by a state machine $sm = \langle i : \mathbb{V}, S : 2^{\mathbb{V}}, trans : 2^{(\mathbb{V} \times MName \times \mathbb{V})} \rangle$, where $trans \subseteq (S \times MName \times S)$ is the relation defining transitions between states and operations of the interface.

We define the translation function for a contract $csp : \mathbb{C}tr \rightarrow \mathbb{C}\textsc{sp}$ by following the outgoing transitions from the initial state of the state machine (denoted in the following by $Ctr.sm$). This also eliminates unreachable parts. Whether we chose a breadth-first or depth-first strategy does not matter. $csp_{sm} : \mathbb{V} \rightarrow \mathbb{C}\textsc{sp}$ recursively translates all outgoing transitions of a state. We map each state in the state machine (italic $s, t, \ldots$) to a CSP process (typewriter $\mathtt{s}$,$\mathtt{t}$,...) using unique names. We remind the reader that these are *variables* and thus placeholders for the identifier of a state or process.

Also, we need to consider that we need slightly different translations depending on the target mode of our translation: at least the state machine may be in the role of specification or implementation (sequence diagram vs. state machine, state machine vs. implementation, two implementations against each other). The specification is always translated using internal choice, while the implementation uses external choice for call-ins. Therefore, we use in the definitions below the choice operator $\prod$ to indicate that we choose angelic or demonic behaviour depending on the context we use the translation in. Thus we have:

$$
\begin{aligned}
&\mathit{csp}\ :\ \mathbb{C}\mathit{tr}_{SM} \to \mathbb{C}\mathrm{SP}\\
&\mathit{csp}(\mathit{Ctr}) = \mathtt{P_{Init}} \cup \mathit{csp}_{sm}(\mathit{Ctr.sm.i}),\\
&\quad \mathtt{P_{Init}} = \prod_{\bar{c}\ \in \mathit{Ctr.Init}(\mathit{Ctr.I.FDec}_{priv})} \mathtt{sm_i}(\bar{c}) \qquad \text{(non-deterministic initialisation of } \mathit{I.FDec}_{priv}\text{)}
\end{aligned}
$$

where $\mathtt{P_{Init}}$ is a CSP process invoking the process corresponding to the initial state $\mathtt{sm_i}$ with given values $\bar{c}$ as per the specification if $I.init$ from the interface for the formal parameters $\overline{st} = I.FDec$, and

$$
\begin{aligned}
&\mathit{csp}_{sm} : \mathbb{V} \to \mathbb{C}\mathrm{SP}\\
&\mathit{csp}_{sm}(s) = \mathtt{s}(\overline{st}) \cup \bigcup_{t \in ts} \mathit{csp}_{sm}(t.t) && \text{(translation of transitive closure)}\\
&\quad \mathtt{s}(\overline{st}) = \mathit{csp}_{trans}(ts, \emptyset) && \text{(a process for each state)}\\
&\qquad ts = \{\langle s,m,t\rangle \mid \langle s,m,t\rangle \in \mathit{sm.trans}\} && \text{(outgoing transitions of } s\text{).}
\end{aligned}
$$

In the translation of the outgoing transitions in csp_{trans}, we choose a translation that returns a single CSP process definition and handles both kinds of choice: as specification, we are allowed to call any method enabled by the protocol (internal choice), as implementation we have to accept every path the environment exercises (external choice). Guards have to be handled specially when translating with internal choice. The specification must only chose enabled courses of interaction, thus we turn each guard into an if-then-else over the guarding expression. A specification may still deadlock if no branch is enabled. The recursive construction below collects all enabled transitions in the specification and implementation and allows them to chose any enabled transition:

$$
\begin{aligned}
&\mathit{csp}_{trans}(ts, ps) =\\
&\begin{cases}
\mathit{csp}_{val}(m.g)\ \&\ \mathit{csp}(\langle s,m,t\rangle) & \text{iff } ts = \{\langle s,m,t\rangle\}, ps = \emptyset\\
(\mathit{csp}(\langle s,m,t\rangle) \prod ps) \lhd \mathit{csp}_{val}(m.g) \rhd (\prod ps) & \text{iff } ts = \{\langle s,m,t\rangle\}, ps \neq \emptyset\\
\mathit{csp}_{trans}(ts \backslash \{t\}, ps \cup \{\mathit{csp}(\langle s,m,t\rangle)\}) \lhd \mathit{csp}_{val}(m.g) \rhd \mathit{csp}_{trans}(ts\backslash\{t\}, ps), \\
\qquad t \in ts,\ \text{otherwise (push choice into branches).}
\end{cases}
\end{aligned}
$$

The function csp_{val} for translating integer and boolean operations into CSP is omitted, since we do not handle navigation paths and method calls as we assume 'flat' rCOS as input. We only allow simple boolean and integer operations (and finite sets thereof). We do not consider the degenerate case of a contract without any methods. For $\mathit{csp}(\langle s,m,t\rangle)$, we handle the translation of a transition to a regular state in CSP:

$$
\mathit{csp}(\langle s,m,t\rangle) = \mathtt{call\_}m?\bar{x} \to \mathit{csp}_m\left(m, \mathtt{ret\_}m!\bar{c} \to \mathtt{t}(\overline{st})\right)
$$

We split the call and return of a method m into two different events $\mathtt{call\_}m$ and $\mathtt{ret\_}m$, so that the environment can synchronize on them. The return values $\bar{c}$ in the output event must be valid expressions over variables in the formal parameters $\overline{st}$ plus any locally declared variables. The process $\mathtt{t}$ is created by the previous definition of csp_{sm} since we translate all states in the transitive closure. Again choice resolves the non-determinism if there is more than one successor state (consider e.g. the predicate $\textsc{exmode}{=}\mathit{true} \sqcap \textsc{exmode}{=}\mathit{false}$ which allows progress with either of the values). Likewise, $csp_{val}(m.g)$ is the abstraction of an rCOS guard into CSP. In case of a deterministic *post*-mapping through $csp_m()$ below, it degenerates into a single branch.

The declaration of the CSP channels for events $\texttt{call\_}m$ and $\texttt{ret\_}m$ requires the mapping of 'flat' rCOS types of input and output/return parameters to CSP data types (currently only integers and boolean, for objects and set-like data structures see future work).

We define the translation of *a sequential composition of designs* mixed with CSP expressions, assuming right-associativity of the operator to simplify the rules: $d_1; d_2; d_3, \ldots = d_1; (d_2; (d_3; (\ldots) \ldots))$. We convert an imperative program with assignments into a functional style, where d_e is the continuation that has to be executed/translated at the end of a sequence:

$$\begin{array}{lll}
csp_m(d, d_e) = & \text{match } d \text{ with} & \\
\mid \textsc{skip} & \longrightarrow csp_m(d_e, \textsc{skip}) \text{ if } d_e \neq \textsc{skip}, & \\
 & \quad \textsc{skip} \qquad\qquad \text{else (end of translation branch)} & \\
\mid d_1; d_2 & \longrightarrow csp_m(d_1, d_2; d_e) & \\
\mid \text{Var } T\ x; d_v; \text{End } x & \longrightarrow csp_m(d_v, \textsc{skip}); csp_m(d_e, \textsc{skip}) & \\
 & \quad \text{(assignments introduce new scope)} & \\
\mid x := e & \longrightarrow \displaystyle\prod_{x \in csp_{val}(e)} csp_m(d_e, \textsc{skip}) & \text{(new scope for } x\text{)} \\
\mid \star(e, d_1) & \longrightarrow \mathtt{W}(\overline{st}), \text{where } \mathtt{W} \text{ is a fresh process name,} & \\
 & \quad \mathtt{W}(\overline{st}) = csp_m(d_1, \mathtt{W}(\overline{st})) \lhd csp_{val}(e) \rhd csp_m(d_e, \textsc{skip}) & \\
\mid [p \vdash R] & \longrightarrow csp_m(R, d_e) \lhd csp_{val}(p) \rhd \mathtt{STOP} & \\
\mid d_1 \lhd e \rhd d_2 & \longrightarrow csp_m(d_1, d_e) \lhd csp_{val}(e) \rhd csp_m(d_2, d_e) & \\
\mid d_1 \sqcap d_2 & \longrightarrow csp_m(d_1, d_e) \prod csp_m(d_2, d_e) & \\
\mid \textsc{chaos} & \longrightarrow \mathtt{CHAOS} & \\
\mid \mathtt{csp} & \longrightarrow \mathtt{csp}; csp_m(d_e, \textsc{skip}) & \text{(any CSP process)}
\end{array}$$

The translation of contracts in $\mathbb{C}tr_{SeqD}$, contracts whose protocol is defined as a sequence diagram, is done by the function $csp : \mathbb{C}tr_{SeqD} \to \mathbb{C}\textsc{sp}$, which proceeds in a similar manner and exercises the same syntactical features of rCOS.

Verification of consistency and implementation As we have shown in Fig. 1, we can now verify by using the FDR2 model checker that given a state machine sm and contract $Ctr_{SM} = (I, Init, MSpec, sm)$ and given a sequence diagram and a contract $Ctr_{SeqD} = (I, Init, MSpec, seqd)$, we have $csp(Ctr_{SeqD}) \underset{M}{\parallel} csp(Ctr_{SM})$ is deadlock free, in which M is the set of generated CSP events (the method calls), that is, the traces specified in the sequence diagram are accepted by the state machine.

Before we can consider verifying an implementation against a contract, we discuss an abstraction framework that can be used to get a suitable over-approximation of an rCOS class into 'flat' rCOS that can be translated into CSP.

Integration In [2], we describe how we integrate the state machine with a class containing the functionality specification to obtain a guarded design that we summarize in the following. We introduce a new variable *state* that holds a symbolic representation of the state that execution is currently in. Then, for each operation m we add a guard that only accepts the call-in into the method if we are in a state that has an outgoing transition labeled with m. Any additional guards on the transition (in the example the test ITEMCOUNTER < MAX) or on the design of m need to be respected as well. In the body, we update the state variable to the successor state.

We do not repeat the formalisation in [2] here, but rather use the example to illustrate the effect on the ENTERITEM method. By integrating the class with the contract as given through the state machine we obtain:

```
public enterItem(Barcode code, int qty ; ) {
  ((state = 13928840) ∨ (state = 9528594) ∨ (state = 14696984) ∨
  ((itemCounter < max) ∧ (state = 4957896))) &
  VAR LineItem item ;
  [ pre  : store.catalog.find(c) != null ,
    post : line' = LineItem.new(c, q) ;
           line.subtotal' = store.catalog.find(c).price ∗ q;
           sale.lines.add(line)]  ;
  [ ⊢ itemCounter' = itemCounter + 1 ] ;
  if (state = 13928840) then {[ ⊢ state' = 9528594]}
  else { if (state = 9528594) then {[ ⊢ state' = 9528594]}
         else { if (state = 14696984) then {[ ⊢ state' = 4957896]}
                else { if (state = 4957896) then {[ ⊢ state' = 4957896]}
                       else { skip /∗ not reached ∗/}}}}}
```

The state identifiers have been automatically generated. Observe how ENTERITEM has been used on four transitions, where one of them held the additional guard. Formally, we denote this integration of a contract and a class by $\mathit{integrate}(\mathit{Ctr}, C)$ (compare with Fig. 1 again).

The Abstraction Process

In order to translate an rCOS specification to a CSP process, it may be necessary to abstract some parts of the designs. Indeed, a design usually contains functional specifications, which may not be relevant to the conformance to the protocol.

The design for ENTERITEM(BARCODE CODE, *int* QTY) above contains information concerning both the protocol and the functional specification. Indeed, the variable ITEM and the first pre/post-condition (checking if the code is in the store catalog and adding the item to the sale) have no concern with the protocol. It is then possible to 'remove' the statements related to ITEM, or, in other words, to keep only those related to the variables concerned with the protocol, which are ITEMCOUNTER, STATE and MAX for ENTERITEM. Moreover, in the perspective of a 'flat' rCOS, all references to object with navigation paths (*i.e.* $x.y$ where $x \neq$ this) should also be removed. We first introduce the function

$\mu : Exp \times 2^{\mathrm{VAR}} \rightarrow \mathbb{B}$, which indicates if an expression should be kept or not:

$$
\begin{array}{lll}
\mu(e,l) = & \text{match } e \text{ with} & \\
\mid \mathit{const} & \rightarrow \mathit{true} & \\
\mid x & \rightarrow \mathit{true} & \text{if } x \in l \\
\mid x & \rightarrow \mathit{false} & \text{if } x \notin l \\
\mid \mathit{path}.x & \rightarrow \mathit{true} & \text{if } x \in l \wedge \mathit{path} = \text{this} \\
\mid \mathit{path}.x & \rightarrow \mathit{false} & \text{if } x \notin l \vee \mathit{path} \neq \text{this} \\
\mid \neg e_1 & \rightarrow \mu(e_1, l) & \\
\mid e_1 \text{ op } e_2 & \rightarrow \mu(e_1,l) \wedge \mu(e_2,l) \text{ (where op } \in \{\wedge, \vee, =, \neq, +, -, /, *\}) & \\
\mid m(in; out) & \rightarrow \mathit{false} &
\end{array}
$$

The abstraction process is an over-approximation, so every entity should refine its abstraction. We introduce the function $\alpha_p : \mathit{Pred} \times 2^{\mathrm{VAR}} \rightarrow \mathit{Pred}$ which abstracts every non atomic expression to *true*.

The designs are abstracted by the function $\alpha_k : D \times 2^{\mathrm{VAR}} \rightarrow D$:

$$
\begin{array}{lll}
\alpha_k(d,l) = & \text{match } d \text{ with} & \\
\mid [p \vdash R] & \rightarrow [\alpha_p(p,l) \vdash \alpha_p(R,l)] & \\
\mid d_1 \lhd e \rhd d_2 & \rightarrow \alpha_k(d_1,l) \lhd e \rhd \alpha_k(d_2,l) & \text{if } \mu(e,l) \\
\mid d_1 \lhd e \rhd d_2 & \rightarrow \alpha_k(d_1,l) \sqcap \alpha_k(d_2,l) & \text{if } \neg\mu(e,l) \\
\mid d_1 \sqcap d_2 & \rightarrow \alpha_k(d_1,l) \sqcap \alpha_k(d_2,l) & \\
\mid d_1 ; d_2 & \rightarrow \alpha_k(d_1,l); \alpha_k(d_2,l) & \\
\mid \star(e, d_1) & \rightarrow \star(e, \alpha_k(d_1,l)) & \text{if } \mu(e,l) \\
\mid \star(e, d_1) & \rightarrow \text{Var bool } b; (b := \mathit{true} \sqcap b := \mathit{false}); & \\
 & \quad \star(b, \alpha_k(d_1,l); (b := \mathit{false} \sqcap b := \mathit{true})) \text{ if } \neg\mu(e,l) & \\
\mid x := e & \rightarrow x := e & \text{if } x \in l \wedge \mu(e,l) \\
\mid x := e & \rightarrow \text{SKIP} & \text{if } x \notin l \vee \neg\mu(e,l) \\
\mid \mathit{path}.x := e & \rightarrow x := e & \text{if } x \in l \wedge \mu(e,l) \wedge \mathit{path} = \text{ this} \\
\mid \mathit{path}.x := e & \rightarrow \text{SKIP} & \text{if } x \notin l \vee \neg\mu(e,l) \vee \mathit{path} \neq \text{ this} \\
\mid \text{Var } T\ x & \rightarrow \text{Var } T\ x & \text{if } x \in l \\
\mid \text{Var } T\ x & \rightarrow \text{SKIP} & \text{if } x \notin l \\
\mid m(in, out) & \rightarrow \text{SKIP} & \\
\mid \text{SKIP}, \text{CHAOS} & \rightarrow \text{SKIP}, \text{CHAOS} &
\end{array}
$$

This function removes every statement not only composed with variables given as a parameter and atomic expressions. Note that the conditional statement is asbtracted as a non deterministic choice if the condition is not atomic. In the same way, if the condition of a loop is not atomic, a new boolean variable b is introduced, and the design of the loop is extended to let the choice to set b to *true* or to *false*.

We extend the abstraction α_k to methods with the function $\alpha_m : \mathit{Meth} \times 2^{\mathrm{VAR}} \rightarrow \mathit{Meth}$ which, given a method and a list of variables, returns the method with the corresponding abstracted design.

Since the above design does not contain any reference to CODE or QTY, we can create a new method by removing these parameters from ENTERITEM. This abstraction is done by the function $\alpha_{par} : \mathit{Meth} \times 2^{\mathrm{VAR}} \to \mathit{Meth}$, defined by:

$$\alpha_{par}((\mathit{Name}, \mathit{in}, \mathit{out}, g, d), l) = (\mathit{Name}, \mathit{in}', \mathit{out}', g, d)$$

with $\mathit{in}' = \mathit{in} \setminus (l \setminus FV(D))$, $\mathit{out}' = \mathit{out} \setminus (l \setminus FV(D))$ and $FV(D)$ stands for the free variables in D. Note by removing only $(l \setminus FV(D))$, we ensure to remove only unused parameters. The functions α_k and α_{par} can be composed to define the function $\alpha : \mathit{Meth} \times 2^{\mathrm{VAR}} \to \mathit{Meth}$:

$$\alpha((\mathit{Name}, \mathit{in}, \mathit{out}, g, d), l) = \alpha_{par}(\alpha_m((\mathit{Name}, \mathit{in}, \mathit{out}, g, d), l), \mathit{in} \cup \mathit{out})$$

The method ENTERITEM can be abstracted by α, by keeping only the variables ITEMCOUNTER, STATE and MAX:

```
α(enterItem, {itemCounter, state, max}) =
public enterItem() {
  ((state = 13928840) ∨ (state = 9528594) ∨ (state = 14696984) ∨
  ((itemCounter < max) ∧ (state = 4957896))) &
  [ ⊢ itemCounter' = itemCounter + 1 ] ;
  if (state = 13928840) then {[ ⊢ state' = 9528594]}
  else { if (state = 9528594) then {[ ⊢ state' = 9528594]}
         else { if (state = 14696984) then {[ ⊢ state' = 4957896]}
                else { if (state = 4957896) then {[ ⊢ state' = 4957896]}
                       else { skip /* not reached */}}}}}
```

Finally, we introduce the function $\alpha_C : C \times 2^{FDec} \to C$ which abstracts all the methods of a given class: $\alpha_C((\mathit{FDec}, \mathit{MDec}), l) = (l, \{\alpha(m, l) \mid m \in \mathit{MDec}\})$, (abstraction w.r.t. variables in l).
Now we are in a position to show that the integrated class still follows a contract: we first calculate the integrated specification from the contract $\mathit{Ctr} \in \mathbb{C}\mathit{tr}_{SM}$ and the implementation C, apply the appropriate abstraction function with respect to the interface to obtain a new flat rCOS specification *impl*. Then, we translate the abstracted methods into CSP and create a process that accepts a call to a method based on the current state through external choice, updates its state variables and starts over:

$$\begin{aligned} \mathit{impl}_{\mathit{Ctr},C} &= \alpha_C(\mathit{integrate}(\mathit{Ctr}, C), \mathit{Ctr}.I.\mathit{FDec}_{priv}) \\ \mathtt{P}_C(\mathit{state}, \overline{st}) &= \mathop{\Box}_{m \in \mathit{impl}.\mathit{MSpec}} \mathit{csp}(m, \mathtt{P}_C((\mathtt{state}, \overline{st}))) \end{aligned}$$

We also need an initial process to start execution and have to state the initial state (derived from the state machine) and the initial values for the remaining attributes which are given in the contract, that is:

$$\mathtt{Init}_C = \prod \left\{ \left(\prod \{ \mathtt{P}(\mathtt{i}, \overline{st}) \mid \overline{st} \in \mathit{Ctr}.\mathit{Init}(\mathit{Ctr}.I.\mathit{FDec}) \} \right) \mid i \in \mathit{Ctr}.\mathit{sm}.i \right\}$$

Then we check the trace refinement property of *impl* against $\mathit{csp}(\mathit{Ctr})$. While the CSP generated from the contract (state machine) consists of a process for each state and a single method call can occur in various places, the overall structure of the CSP for the integrated specification is a single process allowing external choice over all possible guarded method calls, after which it will return to its main process again.

We could also apply the CSP translation of a whole class to an implementation we received from a third party to validate that it respects the contract, provided that we can find a suitable abstraction function.

4 Conclusion

We integrate the power of consistent UML modelling with the application of formal methods: a requirements model of a use case based on the rCOS methodology consists of hierarchical components, a protocol for an interface contract given as a state machine, the scenario to implement in the form of a sequence diagram, and the functionality specification in rCOS. Specification of behaviour in rCOS usually considers trace languages of method invocations instead of data.

The state machines and sequence diagrams can make use of the non-object oriented subset of rCOS designs, that we call 'flat' rCOS, for the functionality specification of operations. This seems like a very strong restriction in the age of object-oriented or even component-based development. But we show based on our example from the recent CoCoME [18] case study for a component-based point-of-sale system that this approach is already sufficient to ensure consistency of the specification between the sequence diagram and the state machine, and the state machine and an implementation (for example, it successfully detects an accidental inconsistency between an older version of the diagrams in [4]). We translate the artifacts into the process algebra CSP and use the model checker FDR2 to check the deadlock freedom. From our limited experience, as any counter example is a sequence of method invocations, it is easy to debug the model with regard to that particular trace. To get from a fully object-oriented rCOS specification of a class to 'flat' rCOS that is suitable for translation to CSP, the user employs a set of abstraction functions, like projection of a class onto a subset of its attributes.

As an illustration, we apply the technique from [2] to obtain an integrated specification in the form of a complete guarded design from the model and check it against the state machine.

We have implemented the rCOS modeler on top of the Eclipse Rich Client platform that can be used to graphically design a requirements model [5] and run the above transformation. Next, we intend to provide different abstractions for sets (e.g. precise, two- or three-valued) and for object instantiation.

In the larger scope, we plan semi-automated model transformation from a requirements model to a design- and component-model [21], proof support for refinement and abstractions, and code generation to provide an integrated software engineering solution for use case-driven models.

The Eclipse plugin of the rCOS modeler and the model used in this paper are freely available from `http://rcos.iist.unu.edu`.

Acknowledgements

This work was partially supported by the projects HighQSoftD and HTTS funded by the Macao Science and Technology Development Fund, by the National Basic Research Program of China (973) under Grant No. 2005CB3218025 and NSFC under Grants No. 90612009, 60803042 and 90718014, and the Grant STCSM No. 08510700300.

References

1. 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.
2. 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, Harrachov, Czech Republic, January 2007. Springer.
3. 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 Rausch et al. [18], chapter 3.
4. Z. Chen, X. Li, Z. Liu, V. Stolz, and L. Yang. Harnessing rCOS for tool support - the CoCoME Experience. In C. B. Jones, Z. Liu, and J. Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 83–114. Springer, 2007. UNU-IIST TR 383.
5. 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.
6. Formal Systems (Europe) Ltd. FDR2 User Manual, 2005.
7. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
8. C. A. R. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.
9. C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice-Hall Intl., 3rd edition, 2005.
10. 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.
11. S. Mellor and M. Balcer. *Executable UML: A foundation for model-driven architecture.* Addison Wesley, 2002.
12. M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Integrating a formal method into a software engineering process with UML and Java. *Formal Aspects of Computing*, 20(2):161–204, March 2008.
13. M. Y. Ng and M. Butler. Tool support for visualizing CSP in UML. In *ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods*, volume 2495, pages 287–298, London, UK, 2002. Springer-Verlag.

14. M. Y. Ng and M. Butler. Towards formalizing UML state diagrams in CSP. In *SEFM*, pages 138–147. IEEE Computer Society, 2003.
15. NoMagic, Inc. Magicdraw. `http://www.magicdraw.com`.
16. Object Management Group. MDA Guide, 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.
17. 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.* Elsevier, April 2008.
18. A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors. *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*. Springer, 2008.
19. A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, 1997.
20. C. F. Snook and M. J. Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.
21. L. Yang and V. Stolz. Integrating refinement into software development tools. In Pu and Stolz [17], pages 69–88.
22. X. Yu, Z. Wang, G. Pu, D. Mao, and J. Liu. The verification of rCOS using Spin. In Pu and Stolz [17], pages 49–67.