SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

December 2014, Vol. 57 128102:1–128102:17
doi: 10.1007/s11432-014-5159-8

Automated transformations from UML behavior models to contracts

LI Dan[1,2*], LI XiaoShan[1], LIU ZhiMing[4] & Volker STOLZ[3]

[1]*Faculty of Science and Technology, University of Macau, Macau, China;*
[2]*Guizhou Academy of Sciences, Guiyang* 550001*, China;*
[3]*Department of Informatics, University of Oslo,* 0316*, Norway;*
[4]*School of Computing, Telecommunications and Networks, Birmingham City University,* B422SU*, UK*

Received November 15, 2013; accepted May 1, 2014

Abstract In model driven architecture (MDA), system requirements are first captured by UML (unified modeling language) use cases with sequence diagrams to describe their intended use and implemented by classes of objected-oriented languages in the subsequent design stages. It is important that the dynamic behavior specified by the sequence diagrams is in full compliance with the implementation classes. This paper proposes an automatic approach and tool support for generating class contracts, which define a precondition and a postcondition for each operation of the class. The former serves as a guard to ensure invocations of the operations respect the semantics introduced by the sequence diagrams, and the latter places the system in a legal state to facilitate the succeeding operation calls. The contracts can be easily mapped to code of an object-oriented language such as Java. Thus, the approach helps to bridge the gap between the requirements and design stages of system development process. We use our model transformation tool to first generate a UML protocol state machine from the sequence diagrams, and then derive the contracts for a controller class. The transformations take into account the concurrency and critical constructs of the respective UML diagrams.

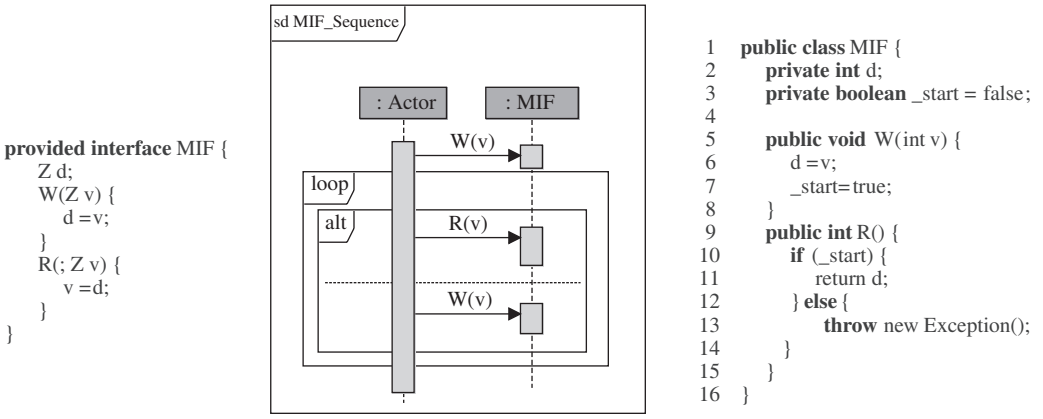
Keywords model-driven development, UML, sequence diagrams, state machines, model transformations, QVT

Citation Li D, Li X S, Liu Z M, et al. Automated transformations from UML behavior models to contracts. Sci China Inf Sci, 2014, 57: 128102(17), doi: 10.1007/s11432-014-5159-8

1 Introduction

Capturing and specifying requirements has been recognized as an essential part of the software system development [1]. The state-of-the-art proposal by the object management group (OMG) for eliciting and analyzing requirements is to use the model-driven development [2, 3] as the methodology and the unified modeling language (UML) [4] as the formalism. In the resulting model of requirements, domain class diagrams are used to represent structural concepts and their relationships, and use cases describe the services provided by the system. Typically, we model a use case as a UML component, while the interface of the component provides methods through which the actors of the use case interact with the component, and the interactions are specified as UML system sequence diagrams [5] involving only a

*Corresponding author (email: lidan_gz@163.com)



(a) (b) (c)

Figure 1 A memory modeled as interface MIF. (a) Interface MIF; (b) interaction protocol of MIF; (c) Java code for MIF.

single actor and the component interface [6]. The system sequence diagram is important in requirement elicitation because it is easy to use and close to users' understanding.

In the subsequent analysis and design stages, UML state machines are defined as reactive protocols for modeling the reactive behavior of the interface. A state machine is essentially executable and can be validated and tested through simulated executions. Also, the interface, along with its protocols, is realized by a controller class (use case controller) that is responsible for receiving or handling system events [5]. The controller class is then used for object-oriented design, and finally implemented as executable program code. Thus, it is important that the controller classes, as well as their implementations, fully comply with the semantics introduced by the original requirements model.

Example 1. The model shown in Figure 1(a) and 1(b) describes the behavior of a memory specified as interface MIF. It provides two methods W and R, for writing a value to and reading the content out of the memory cell d of type Z. MIF also defines local data functionalities for the two methods as $d = v$ and $v = d$, respectively. Figure 1(b) specifies the interaction protocol for MIF, requiring that the first operation has to be a writing operation.

It is simple and straightforward to implement the structure and data functionalities of the interface with a Java class. However, implementation of the behavioral protocol is relatively difficult, as Java does not directly support this mechanism. One solution is to introduce state variables through an intermediate step of designing a state machine for the protocol. Figure 1(c) shows a Java implementation of the MIF, where a variable _start and an if statement (lines 3 and 10) are introduced to ensure the reading of d can only occur in the condition that _start is true. Line 7 sets the state variable to true to allow the invocation of method R.

Manually designing the state machines and the implementation classes to support the behavioral protocols, especially for complex systems, is difficult and error-prone. Automation of this process provides considerable help for the users. It allows the designers who are not very technical to concentrate on designing the local data functionalities of methods and using sequence diagrams to express high-level interaction requirements. They do not need to worry about how to specify the allowable execution conditions for the methods in the implementation classes. Thus, it helps to bridge the gap between the functional requirements and the design that will satisfy the requirements [5, 7].

In this paper, we present an approach that first transforms the sequence diagrams of a use case to a UML protocol state machine, based on which a controller class is then derived. The contracts of the generated class include a pre- and a postcondition for each operation that triggers transitions in the state machine. The precondition serves as a guard to ensure invocations of the operations follow the semantics introduced by the sequence diagrams, and the postcondition places the system in a suitable state on which only allowable operations can be called. From the contracts, we can directly generate Java classes and method skeletons and fill in the skeletons with implementations of the local data functionalities (such as

lines 6 and 11 in Figure 1), which are manually implemented or generated by other tools. The contracts can also be further translated to, for instance, Java Modeling Language (JML) [8] scripts and annotated to corresponding Java classes. JML is used for runtime assertions that reveal if a particular execution of the system breaks the precondition or postcondition of a method. The assertions are automated with existing tools. In addition, the generated protocol state machines can be used to analyze and check the system.

We have implemented the approach as a tool, which automates the transformations of protocol state machines from sequence diagrams and the generation of controller classes with method contracts. We integrated the tool into rCOS modeler, an Eclipse-based CASE tool for rCOS method [6]. The transformations are defined in the graphical notation of QVT relations language (QVT-R) [9] using our QVTR-XSLT tool [10]. A range of interaction operators of sequence diagram, such as alt, opt, seq, neg, loop, and break, are considered in the transformations. In particular, we support par and critical, which provide the ability to specify concurrent and safety critical requirements. Using the interaction uses that refer to other sequence diagrams, we can elaborate the scenarios of a complex use case by a set of sequence diagrams and describe their relationships with a sequence diagram at a higher level. Also, the tool can generate the state machines with graphical representations, so the users are able to immediately inspect the transformation results visually.

The rest of this paper is organized as follows. In Section 2, we introduce the basic concepts of the models for the transformations. Section 3 presents the main contribution — the major steps of the transformations. We discuss transformation implementation and tool design in Section 4. Section 5 presents a case study. We compare the approach with related work in Section 6 and summarize the conclusions in Section 7.

2 Models for transformations

This section introduces the UML artifacts we deal with: classes, interfaces, sequence diagrams, and protocol state machines. Our work is based on a subset of standard UML 2 proposed by OMG [4].

2.1 Class and interface

As one of the most important UML diagrams, a class diagram is used to model the static structure of a system. Within the class diagram, we can define domain concepts as classes and their relationships. Meanwhile, a UML interface defines a contract that represents a declaration of a set of coherent public features and obligations [4]. Both class and interface contain properties and operations whose functionalities are specified by the pre- and postconditions. An operation can be a query that does not change the system state.

In our model, an interface is realized by a controller class, which is an artificial class (not one in the domain class model) specially created for the whole system or for each use case. A controller class can act as an entry point of a system. It is responsible for initializing the system, creating permanent objects of the system, keeping the main flow of control of the system, and receiving or handling system events [11].

2.2 System sequence diagram

UML sequence diagram (SD) provides powerful and well-defined constructs to rigorously model complex flows of control. A sequence diagram includes:

• a set of lifelines, each representing a participant taking part in the interaction. For a system sequence diagram, there are only two lifelines, one represents an actor and the other represents an interface.

• a set of messages, each specifying a communication from a sender lifeline to a receiver lifeline through a pair of send and receive events and corresponds to an invocation of an operation. In our model, we allow only synchronous messages calling to operations defined in the interfaces.

• an ordered set of interaction fragments, each representing a piece of the interaction. The fragments cover lifelines from top to bottom in a time sequence. There are three kinds of interaction fragments:

– message occurrence: specifies the occurrence of the pair of events caused by an operation call.

– combined fragment: expresses varied kinds of conditional control flows. A combined fragment consists of one or more interaction operands, which may also contain message occurrences and combined fragments. An interaction operand can own a constraint as its guard condition. The type of a combined fragment is decided by its interaction operator.

– interaction use: a shorthand for copying the contents of the referred interaction to where the interaction use is. The referred interaction is usually defined in its own sequence diagram. Interaction uses allow reuse of existing sequence diagrams and decomposition of a complex sequence diagram into simpler ones.

The behavioral semantics of a sequence diagram can be described as a set of valid traces and a set of invalid traces (see Sec. 14.3.11 of [4]). Here, a trace is a sequence of event occurrences. In our model, a trace can be simplified to a sequence of calls to the operations defined in the interface.

2.3 Protocol state machine

A UML protocol state machine (PSM) is used to specify the usage protocol as the allowed call sequences on the operations of an interface. A distinctive feature of PSM is that a transition is associated with a precondition (guard) and a postcondition, but not with any action, and internal transition triggers cannot be used [4]. Thus, a PSM describes the offered services and not the way they could be implemented. A PSM includes:

• A set of vertices which are abstractions of nodes in a state machine. A vertex may either be a pseudostate or a state, and it has incoming and outgoing transitions. If a state contains regions, it is called a composite state, otherwise it is a simple state. In our case, a composite state is always an orthogonal state, which means there are more than one region inside the state. These regions are called orthogonal regions, and they execute concurrently with interleaving semantics. For pseudostates, we support junction and choice. For convenience, if vertex A has an outgoing transition to vertex B, we call B the outgoing vertex of A. Conversely, A is the incoming vertex of B.

• A set of protocol transitions, each representing a directed relationship between a source vertex and a target vertex. A transition represents a single atomic communication event. It may have a pre- and a postcondition, and it may be triggered by a referred operation that changes the system state. A transition is enabled if its precondition evaluates to true and its source state is active. A transition without precondition, postcondition, and referred operation is a tau transition, which can be executed automatically and does not change the system state.

• A set of regions which contain vertices and transitions. For each region, an initial pseudostate defines its starting point, and a final state signifies the completion of the region. A transition to a composite state means the transitions to each initial pseudostate of its regions, and all the orthogonal regions are activated at the same time, and each region has its own thread of execution. Completion of the orthogonal regions represents completion of the enclosing composite state. A final state in the region directly owned by the state machine signifies the completion of the state machine.

The semantics for PSM, as described in the UML specification, is run-to-completion: a transition completes only after the completion of its referred operation. Thus, while executing an operation, no other operation calls can be processed by the PSM. The semantics is captured by a set of traces of operation calls, each starting from the initial state, going along the outgoing transitions, and to the final state of the PSM.

3 Transformations from requirements models to contracts

The transformations start from a requirements model. It consists of component interfaces that represent use cases, system sequence diagrams which describe the interactions between the actors and the interfaces,

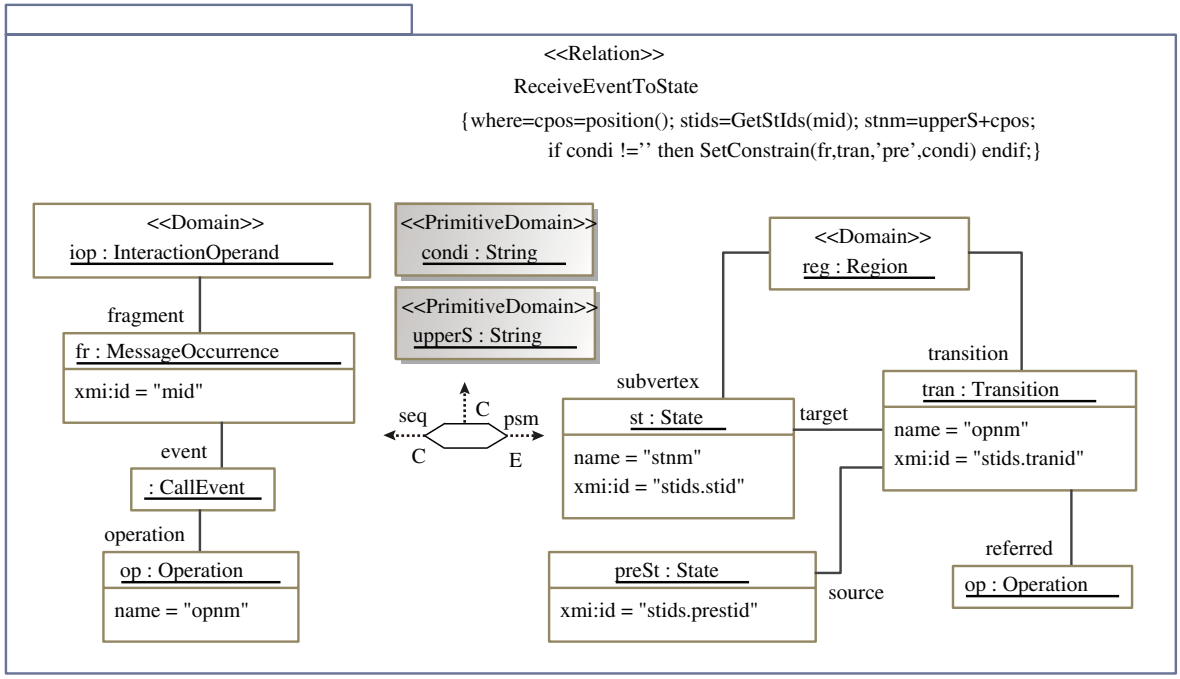


Figure 2 QVT-R rule for message occurrence.

and domain class diagrams for specifying the static structure of the system. Our approach consists of the following transformation steps: (1) first, we synthesize a protocol state machine for an interface from its system sequence diagrams; (2) then we label the protocol state machine by attaching every vertex with state and critical tokens; (3) and this is followed by the generation of a controller class which implements the interface and the labeled protocol state machine.

The transformations are specified with the graphical notation of QVT-R, a standard transformation language proposed by OMG. It provides a concise, intuitive, and yet powerful way to define model transformations. In QVT-R, a transformation is defined as a set of relations (rules) between source and target metamodels, where a relation specifies how two object diagrams, called domain patterns, relate to each other. Optionally, a relation may have a pair of when- and where-clauses specified with object constraint language (OCL) to define the pre- and postconditions of the relation, respectively. A transformation also includes queries and functions.

3.1 From sequence diagrams to protocol state machines

The transformation requires that: given a system sequence diagram SD, and a generated PSM, a call event to an operation of the interface in the SD becomes a transition of the PSM that is triggered by the same operation, and all valid traces of the SD, after restriction to calling of non-query operations, are the traces of the PSM. The transformation is performed by taking input a requirements model, indicating as argument the lifeline which represents the interface we want to translate, and then producing a protocol state machine and its graphical representation. In the following, we discuss how to deal with different kinds of interaction fragments.

3.1.1 *Message occurrences*

For the message occurrence of a receiving event, the relation shown in Figure 2 is applied. The source domain pattern (left) of the relation matches a message occurrence fr with a call event and a connected operation op. As parameters of the relation, the primitive domains condi and upperS are the guard condition and the name of the parent state, respectively. In the where clause, a unique name stnm is

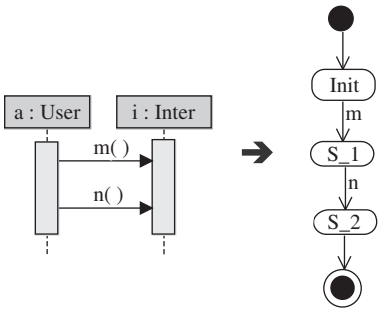


Figure 3 Applying the rule for message occurrence.

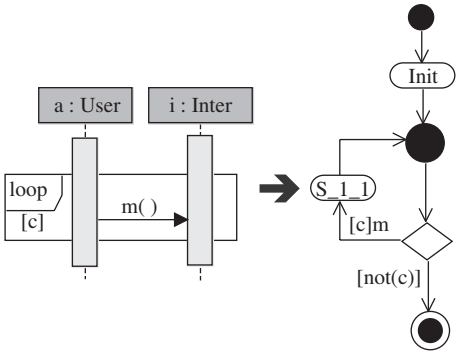


Figure 4 Applying rule for loop.

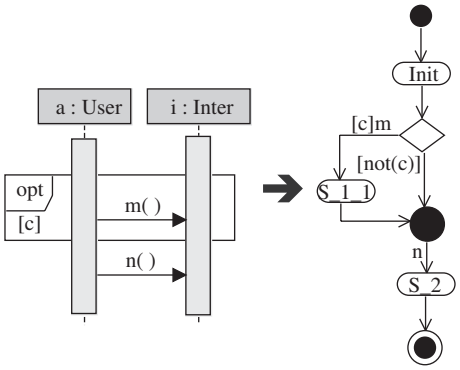


Figure 5 Applying rule for opt.

defined as upperS + cpos, where cpos is the position of fr in the list of all current fragments. Then, the target domain pattern (right) constructs a new state st, assigns its name to stnm, and adds a transition tran from the previous state to the state st. The transition refers to operation op and attaches a precondition if there is one by invoking relation SetConstrain in the where clause. This straightforward strategy ensures that all traces of a SD are also traces of the generated PSM. Figure 3 shows the transformation of a simple sequence diagram by applying this relation. Since UML does not allow that a transition referred to an operation starts from the initial pseudostate, we add a state with name Init before the state S_1.

3.1.2 *Combined fragments*

A combined fragment contains one or more interaction operands, which in turn may include more message occurrences and other combined fragments. After transforming the combined fragment itself, the contents of each operand need to be recursively processed. The transformation rules for the combined fragments are quite similar. We therefore do not dwell on the rules themselves but merely use examples to illustrate the translation results.

• **alt**, **break**, **loop**, **opt**: These interaction operators are used to designate a choice of behavior that is controlled by a guard constraints. For example, a loop represents the iterative application of its operand until the guard evaluates to false, and the alt operator allows the modeling of the classic "if-then-else" logic. We generate for each of them a pair of pseudostates (junction or choice) to denote the starting and finishing of the construct along with transitions between the two pseudostates to represent the control flows of the combined fragment. Figures 4 and 5 show the translations of operator loop and opt, respectively. In the diagrams of the generated state machines, a larger solid circle represents a junction pseudostate, a smaller solid circle represents an initial pseudostate, and a choice is represented as a diamond-shaped box.

• **par**, **critical**: The par operator is used to model a parallel composition of its operands. Each operand represents a thread of execution done in parallel. The behaviors of the operands can be interleaved in any order as long as the ordering imposed within each operand is preserved. The critical operator is closely related to the par fragment. It defines a critical area in which the traces of execution cannot be

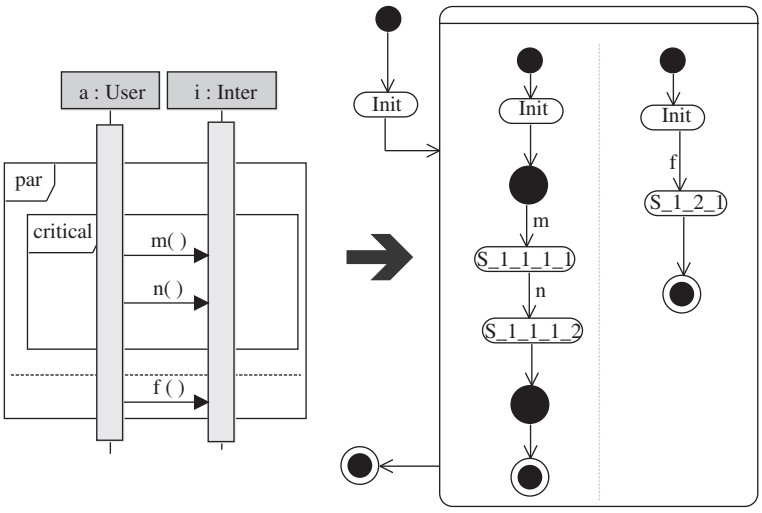


Figure 6 Applying rules for par and critical.

interleaved by other messages. The critical operator is important in modeling a sequence of behaviors that must be contiguously handled.

Figure 6 gives an example of a par fragment with a critical area. A par fragment leads to a composition state containing two or more orthogonal regions. This preserves the traces of the par-block because the UML semantics for orthogonal regions is interleaving of events from different regions. Inside a region, we generate a state machine for each operand of the par. These state machines each have an initial pseudostate and a final pseudostate. All state machines of the orthogonal regions reaching the final states activate the transition leaving the composite state. For a critical fragment, we generate two junction pseudostates to mark the start and end of the critical area. These pseudostates will be labeled with critical tokens in the subsequent step.

• **seq**, **neg**: As we generate one PSM per lifeline, the sequence of actions is already decided by the order of events along the lifeline, so the seq operator does not bring any new semantics. We simply add two junction pseudostates to indicate the start and end points. For the neg fragment, we just ignore the operator and its content. This is because negative behavior shall not be implemented by the state machine. We only map the positive behaviors of a sequence diagram to the protocol state machine.

3.1.3 *Interaction uses*

An interaction use is shown as a combined fragment where the operator is called ref. It can refer to an interaction which in turn may contain other interaction uses, and this process can be recursive. However, a ref cannot contain an interaction use which refers to the interaction itself to avoid infinite loop. As shown in Figure 7, a pair of junction pseudostates is generated to indicate the start and end of the interaction use, and then the referred interaction is subsequently processed.

Redundant constructs of the generated PSM, for example, the two adjacent junction pseudostates in Figure 7, can be optimized to make the PSM more concise (and better readable for humans).

3.2 Labeling protocol state machines

From the generated PSM, we are going to construct the specification for the controller class. Therefore, we have to calculate the conditions for triggering a transition, including its source state, precondition, and the critical constraints. Since UML allows a model element to be associated with a set of constraints, we use this to attach this information to the vertices.

We introduce a state variable of type, for simplicity, string, for each region of the PSM to indicate the current state of the region. The state variable of the uppermost region by default is named "state" (actually "_state" in the model). In addition, we introduce a critical variable that holds a boolean value {ON,OFF} for each region generated from a par fragment with critical areas.

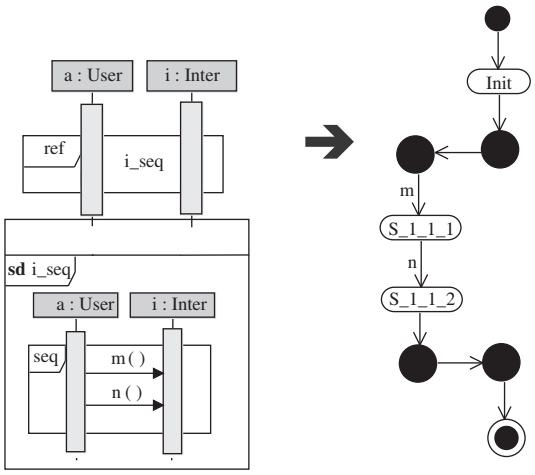


Figure 7 Applying rule for ref.

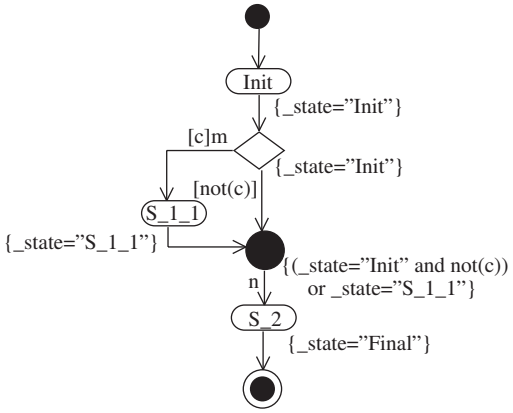


Figure 8 Labeled PSM from Figure 5.

To label the protocol state machines, we define a QVT-R in-place transformation in which both source and target metamodels are the same PSM. This labeling process is carried out by traversing the PSM to label each vertex with tokens referred to the state and critical variables. In the following, we discuss the rules for labeling constructs of the PSM using examples.

• **Labeling simple states**: A simple state is labeled with a token that sets its region's state variable to the name of the state. For example, the state S_1_1 in Figure 8 is labeled with state="S_1_1".

• **Labeling pseudostates**: If all its incoming vertices are labeled, we give a pseudostate a state token that is a disjunction of the predicates from the incoming vertices. Each predicate is a conjunction of the incoming transition's precondition and the incoming vertex's state label. As shown in Figure 8, the junction state is labeled as (state="Init" and not(c)) or state="S_1_1", where c is the guard condition of the opt fragment in Figure 5.

• **Labeling composite states**: A composite state is provided with a prestate token that represents the enter condition, and a state token which represents the exit condition. The prestate token is labeled in the same way as the pseudostates, and the state token indicates all its contained regions reach the final states. Figure 9 shows an example of the labeled composite state.

• **Critical labeling**: If a composition state is generated from a par fragment with critical areas, critical tokens are given to the regions and related states:

– **Region labeling**: Each region is given a critical label that indicates the critical variables of all other orthogonal regions should be OFF. For example, as shown in Figure 9, the region_1_1 is labeled with a critical token critical_1_2=OFF.

– **State labeling**: Recall that a critical area in a *SD* becomes a pair of junction pseudostates in the PSM. We label the outgoing vertex of the starting junction pseudostate with a token to turn on the critical variable of the region, and the incoming vertex of the finishing junction pseudostate a token to turn off the critical variable. As shown in Figure 9, the state S_1_1_1_1 is labeled with critical_1_1=ON and the state S_1_1_1_2 with critical_1_1=OFF.

– **Final state labeling**: If there is a tau transition from a vertex to the final state, we label the vertex with state="Final", cf. state S_2 in Figure 8.

By recursively applying the above rules, a PSM is gradually transformed into a labeled protocol state machine. The transformation completes when no more rules can be applied.

3.3 Generation of controller classes

Given an interface I and its PSM, an implementing controller class C meets the following requirements:

• an operation of I becomes an operation of C, with the same signature and data functionality definition (if any), and the operation of C may contain additional pre- and post-constraints;

• a property of primitive type becomes a property of same type in C;

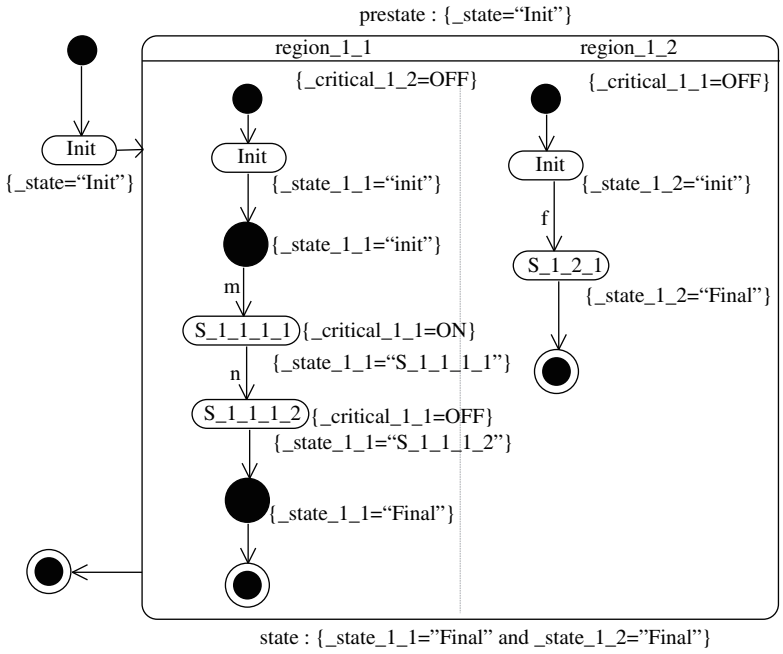


Figure 9 Labeled PSM from Figure 6.

• a property of class type becomes an association from C to that class, and the name of the association is the same as the property's;

• all initial constraints and invariants of I become those of C;

• all traces of the PSM can be executed in C;

• any trace, if it cannot be executed in the PSM, must be blocked in C.

We define another QVT-R transformation for the generation of controller classes. However, implementing the static structure of I in C is straightforward. In the subsection, we focus on the implementation of the dynamic behavior defined in the protocols. Taken as input an interface with a labeled protocol state machine, the transformation generates initial constraints and adds pre- and postconditions to the operations.

3.3.1 *Adding initial constraints*

The state and critical variables introduced in the labeling process become attributes of the controller class, and they have to be initialized when the class is instantiated. For example, the following initial constraints derived from Figure 9 are added to the class declaration:

state="Init"; state_1_1="Init"; state_1_2="Init"; critical_1_2=OFF; critical_1_1=OFF;

3.3.2 *Deriving comprehensive pre- and postconditions for transitions*

A transition may already hold a precondition representing the guard of the corresponding operand. To obtain the comprehensive pre- and postconditions for the transition, we have to consider its source and target vertices. For the precondition, we include the state tokens on the source vertex and all its parent states, as well as the critical tokens for the source vertex and all its parent regions. For deriving postcondition from the target vertex, we consider only the tokens on the vertex itself, since we assume that the execution of the referred operation will not change the state and critical variables. For example, the following pre- and postconditions are generated for transition m in Figure 9:

pre: (state_1_1="Init" **and** critical_1_2=OFF) **and** state="Init";

post: state_1_1="S_1_1_1_1" **and** critical_1_1=ON;

While the pre- and postconditions for transition f are:

pre: (state_1_2="Init" **and** critical_1_1=OFF) **and** state="Init";

post: state_1_2="Final";

Table 1 Transformations and generated XSLT code.

Transformation	No. of relations	No. of queries	No. of functions	Lines of generated XSLT code
SD to PSM	44	21	19	3489
Labeling PSM	29	26	3	1286
Interface & PSM to controller class	19	18	3	2228
Total	92	65	25	7013

It is obvious that the invocation of f cannot immediately follow the invocation of m, as the variable critical_1_1 has been turned on in the postcondition of m. This preserves the semantics of the critical fragment defined in the sequence diagram of Figure 6.

3.3.3 *Generating pre- and postconditions for operations*

In this step, we compose the individual pre- and postconditions of transitions referred to the same operation together to obtain a complete specification for the operation. The precondition of the operation is the disjunction of all transition's preconditions, and we use "if-then-else" expressions to compose the postconditions of the transitions. For example, let m_1 and m_2 be two transitions of operation m, pr_1, pr_2 and pst_1 and pst_2 be the pre- and postconditions of the two transitions, respectively. For operation m in I, we denote the pre- and postconditions of its functionality definition by p and R, respectively. We get the following specification for operation m of C:

m() { **pre**: (pr_1 **or** pr_2) **and** p;
post: **if** pr_1 **then** (pst_1 **and** R) else if pr_2 **then** (pst_2 **and** R) **else** **skip**;}

4 Tool implementation

The transformations described in this paper are implemented in the QVTR-XSLT tool [10]. The tool supports the graphical notation of QVT-R and the execution of a subset of QVT-R by means of XSLT [12]. It provides a graphical editor in which a transformation can be specified using the graphical syntax, and a code generator that automatically generates executable XSLT programs for the transformation.

Using the QVTR-XSLT tool, the three transformations are specified as three transformation models that contain source & target metamodels and transformations which consist of relations, queries, and functions, including a set of relations that produce the corresponding visual presentations for the generated PSM. Table 1 gives the number of relations/queries/functions and lines of generated XSLT code for each transformation. The generated XSLT programs can be directly executed under a XSLT processor, such as Saxon or Xalan, and subsequently transform a requirements model, which is stored as a pair of XML files in the formats of EMF (for UML semantic model) and DI (for visual representation), into controller classes with contracts. However, it is boring and repetitive to execute a transformation in this way. For the convenience of the users, we have integrated the XSLT programs into rCOS modeler, a CASE tool for formal model construction, analysis, and verification. The rCOS modeler provides a UML-like multi-view and multi-notational modeling and design platform with the ability to add plug-ins. It is implemented on top of the Eclipse platform for ensuring compatibility with other UML-based software engineering tools. Topcased is used to support graphic design of models. We develop a set of Java classes to invoke the XSLT programs with necessary arguments. The Java classes are plugged into the rCOS modeler.

With the graphical interface of rCOS modeler, a user can first design a requirements model that consists of component interfaces to represent the use cases, and system sequence diagrams to describe the interactions of the interfaces. As shown in Figure 10, the user can then select an interface lifeline from a sequence diagram and invokes the SD to PSM transformation from a popup menu. She can immediately check and label the generated state machine in the view of state diagram. Then from the view of the component diagram, the transformation of generating the controller class can be invoked. The user can

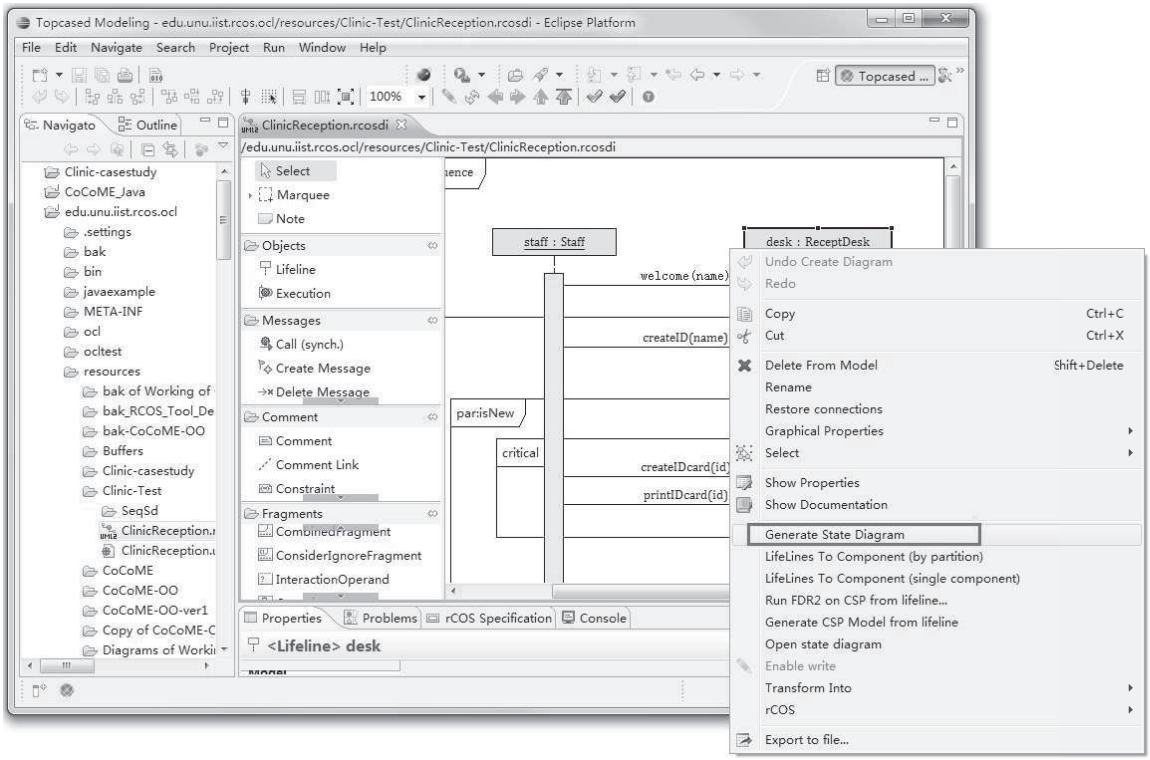


Figure 10 Generate state diagram by invoking the SD to PSM transformation.

inspect the generated class in the view of class diagram or outputs a textual specification of the class definitions. The example diagrams for the paper, except the ones defining the metamodels and QVT-R rules, are the results of those transformations after minimal visual tidy up.

Also, we rely on rCOS modeler to check the correctness of the transformations by ensuring the input and output models are consistent and syntactically correct. The rCOS modeler provides a set of tools for model validation and verification. First, a set of OCL rules is integrated into the modeler to check the completeness and static consistency of different diagrams of a model. Then, we check the dynamic consistency of the model: the state machine must accept all interaction sequences described by the system sequence diagram, and any implementation of the interface must not be deadlock if invoked according to the protocol given through the state machine [13]. This is done by translating the sequence diagram, the state machine, and the controller class into CSP (Communicating Sequential Processes) processes [14] and using the CSP model checker FDR2 [15] to check the deadlock freedom of these CSP models against each other. The translation and checking processes are also fully automatic in the rCOS tool. More details of the validation and verification can be found in [13].

5 Case study

The case study describes the patient reception process at a clinic. A patient arrives at the clinic and stops first at the reception desk. The staff here welcomes the patient and checks if he is recorded in the clinic's system (with key information such as name). For a new patient, the staff creates a unique patient ID, an ID card, and a label showing the waiting number. Both the card and the label need to be printed out, and the printing of the ID card must immediately follow the creation of the card. With the ID card and the label, the patient is sent to a specific doctor's waiting queue. If the patient has visited the clinic before, the staff only needs to produce a label, print it out, and send the patient to the waiting queue.

As shown in Figure 11, the case study is represented as use case Reception and modeled by a component with interface ReceptDesk. The left-hand side of Figure 12 defines the properties and operation signatures

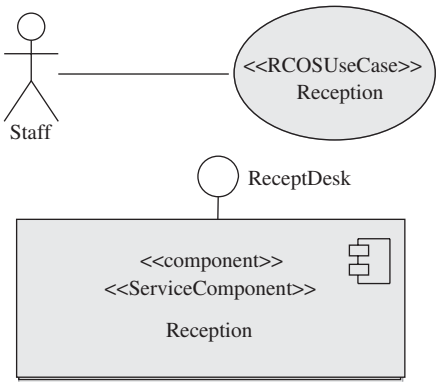


Figure 11 Use case and component.

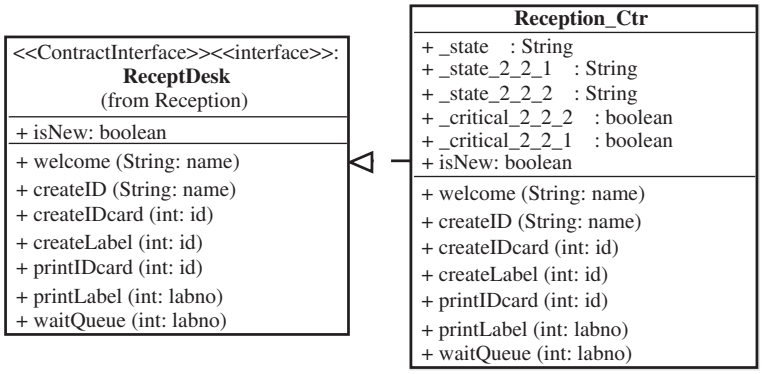


Figure 12 Interface and generated controller class.

of the interface, and we also specify local data functionalities of the operations. For example, the operation welcome is defined as:

welcome(name : **string**) {
if found(name) **then** isNew=**false else** isNew=**true**; }

where found is an abstract function to check whether the patient is in the clinic's database. We omit all other data functionality definitions to save space.

The system sequence diagram shown in Figure 13 describes the scenario of the use case. The isNew condition of the alt fragment determines whether the patient is a newcomer. We use a par fragment to express that the staff can create either the ID card or the label first. Both of them need to be printed. A critical operator is adopted to ensure that the ID card is printed immediately after its creation.

Applying the transformations discussed previously, we first transform the sequence diagram into the PSM illustrated in Figure 14. After labeling it, we generate the controller class Reception_Ctr. The right-hand side of Figure 12 depicts the structure of the class, while the class initializing constraints and the operation specifications are listed below in Figure 15.

The controller class can be further mapped to a Java class in a simple and straightforward way. Figure 16 shows an excerpt of a possible Java implementation, where Java assert statement provides an easy means to check the preconditions and throws an exception if the checking fails.

Validation. An implementation system must follow the semantics introduced by the sequence diagram shown in Figure 13. We exemplify the following properties that the system has to satisfy:

• ***P***1: Operation welcome must be the first operation executed after initialization of the system.

• ***P***2: For a new patient, both operations createIDcard and createLabel should be executed, and they can be invoked in any order.

• ***P***3: The invocation of operation printIDcard must immediately follow the execution of operation createIDcard.

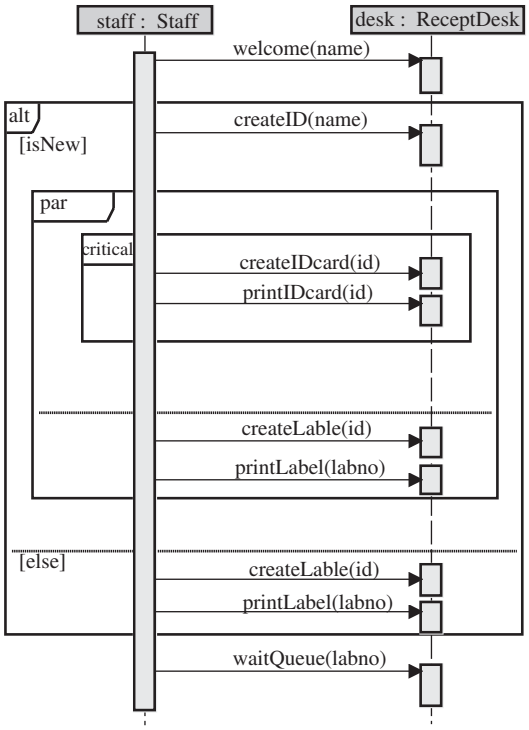


Figure 13 System sequence diagram.

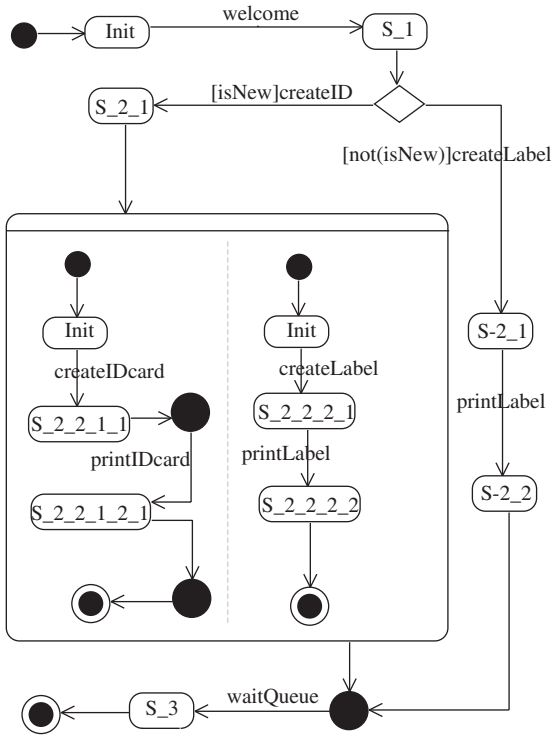


Figure 14 Generated protocol state machine.

Initial: _state="Init", _state_2_2_1="Init", _state_2_2_2="Init", _critical_2_2_2=OFF, _critical_2_2_1=OFF;

welcome(name:String) {
  pre: _state="Init";
  post: (**if** found(name) **then** isNew=**false else** isNew=**true**) **and** _state="S_1";
}
createID(name:String) {
  pre: isNew **and** _state="S_1";
  post: _state="S_2_1" **and** _state_2_2_1="Init" **and** _state_2_2_2="Init";
}
createIDcard(id:**int**) {
  pre: _state_2_2_1="Init" **and** _critical_2_2_2=OFF **and** _state="S_2_1";
  post: _state_2_2_1="S_2_2_1_1" **and** _critical_2_2_1=ON;
}
createLabel(id:**int**) {
  pre: (_state_2_2_2="Init" **and** _critical_2_2_1=OFF **and** _state="S_2_1") **or** (not(isNew) **and** _state="S_1");
  post: **if** (_state_2_2_2="Init" **and** _critical_2_2_1=OFF **and** _state="S_2_1") **then** _state_2_2_2="S_2_2_2_1"
      else if (not(isNew) **and** _state="S_1") **then** _state="S−2_1" **else skip**;
}
printIDcard(id:**int**) {
  pre: _state_2_2_1="S_2_2_1_1" **and** _critical_2_2_2=OFF **and** _state="S_2_1";
  post: _state_2_2_1="Final" **and** _critical_2_2_1=OFF;
}
printLabel(labno:**int**) {
  pre: (_state_2_2_2="S_2_2_2_1" **and** _critical_2_2_1=OFF **and** _state="S_2_1") **or** _state="S−2_1";
  post: **if** (_state_2_2_2="S_2_2_2_1" **and** _critical_2_2_1=OFF **and** _state="S_2_1") **then** _state_2_2_2="Final"
      else if _state="S−2_1" **then** _state="S−2_2" **else skip**;
}
waitQueue(labno:**int**) {
  pre: (_state_2_2_1="Final" **and** _state_2_2_2="Final") **or** _state="S−2_2";
  post: **if** (_state_2_2_1="Final" **and** _state_2_2_2="Final") **or** _state="S−2_2" **then** _state="Final" **else skip**;
}

Figure 15 Method contracts for controller class Reception_Ctr.

```
public class Reception {
    static final boolean ON = true, OFF = false;
    private String _state="Init", _state_2_2_1="Init", _state_2_2_2="Init";
    private boolean _critical_2_2_2=OFF, _critical_2_2_1=OFF; isNew;

    public void welcome(String name) {
        assert _state=="Init"; // precondition
        if (found(name)) { isNew=false; } else { isNew=true; } //data functionality
        if (_state=="Init"){_state="S_1";} //postcondition
    }
    .....
    public void printLabel(int labno) {
        // precondition
        assert (_state_2_2_2=="S_2_2_2_1" && _critical_2_2_1==OFF && _state=="S_2_1") || _state=="S–2_1";
        ..... //data functionality
        //postcondition
        if (_state_2_2_2=="S_2_2_2_1" && _critical_2_2_1==OFF && _state=="S_2_1"){
            _state_2_2_2="Final";
        } else { if (_state=="S–2_1") {_state="S–2_2";}}
    }
}
```

Figure 16 An excerpt of a Java implementation for controller class Reception_Ctr.

• $\boldsymbol{P}4$: Operation printLabel cannot be invoked before the execution of operation createLabel.

• $\boldsymbol{P}5$: Operation waitQueue can only be invoked after the executions of both operations printLabel and printIDcard.

Let $\mathcal{S}$ be an object of Java class Reception shown in Figure 16, then:

• The instantiation of $\mathcal{S}$ makes _state="Init", thus only the precondition of operation welcome is satisfied and the operation can be invoked. Property $\boldsymbol{P}1$ is held.

• After the execution of createID, $\mathcal{S}$ is in a state where both the preconditions of createIDcard and createLabel are satisfied, and each may be invoked. Thus, property $\boldsymbol{P}2$ holds.

• If the createIDcard is executed, $\mathcal{S}$ is in a state of critical_2_2_1=ON, where neither the createLabel nor the printLabel can be invoked until the execution of printIDcard, which turns the critical variable critical_2_2_1 to OFF. Therefore, property $\boldsymbol{P}3$ is satisfied.

• Execution of the printLabel makes $\mathcal{S}$ in a state of _state="S-2_2", in which the precondition of createLabel is not satisfied. Thus, property $\boldsymbol{P}4$ holds.

• The precondition of waitQueue requires $\mathcal{S}$ in a state of state_2_2_1="Final" and state_2_2_2="Final", that can only be met after both the printLabel and the printIDcard are executed. Property $\boldsymbol{P}5$ therefore holds.

Thus, we can say $\mathcal{S}$ satisfies all identified properties, and the behavior introduced by the sequence diagram of Figure 13 is followed by the Java implementation and the controller class.

6 Related work

The transformation from scenario-based interaction diagrams to state-based models is one of the key activities in OOAD [16]. It allows designers to concentrate on modeling system requirements and automatically get executable and simulatable state models which are required in design processes. Algorithms that transform scenarios into state machines are often called synthesis algorithms. Many approaches have been proposed to address the algorithms. Amyot and Eberlein [17] and Liang et al. [18] have evaluated more than 20 of them. However, most of the works date back before UML 2.0. The scenario languages they used lack control constructs, such as combined fragments and interaction uses introduced in UML 2.0 sequence diagrams. Scenarios are written in isolation and their relationships are not specified [19]. Therefore, the problem that mainly captured researcher's attention is how to decide the relationships between different scenarios. These approaches require a significant additional modeling effort from the users [19].

The work of [20] deals with most combined fragments of UML 2.0, such as alt, opt, loop, and par, but leaves out the interaction uses. The transformation is defined through graph transformations based on concrete syntax. In [21], an algebraic framework for synthesizing statecharts from UML 2.0 sequence diagrams was proposed. Only three operators (seq, alt, and loop) are supported, and the approach takes sequence diagrams in textual format as inputs. The aforementioned [19] uses interaction overview diagrams, instead of interaction uses, to specify relationships between scenarios. This is the only work we have seen that supports the interaction overview diagrams. Most work in this area focus on the synthesis algorithms, whereas the integration in industrial practice remains implicit, and many of the approaches are not supported by tools [22, 23].

Our SD to PSM transformation is similar to the ones described in [19, 20] in that we support most of the combined fragments of UML 2.0, including the interaction uses. Particularly we translate parallel into PSM's orthogonal regions, and we support critical fragments, which we think is important in terms of defining critical applications, but never mentioned in the literature about synthesis. We integrate multiple scenarios of a single interface through interaction uses. Our transformation generates PSM suitable for the definition of contract protocols.

Our approach to label the state machines is similar to the techniques used in [24], where state variables are introduced to represent the current configuration. The system controls a single variable which represents the message sent by a system object in the current step. Then, the current state of the system automaton can be represented by the variable, ranging over its states. In our approach, we define a state variable for each region of the PSM and simply use the name of a state as the value of the state variable. However, the algorithm we used to label the state machines is unique as we consider hierarchical state and critical variables. The work of [25] also generates class contracts, as a pre- and a postcondition for each operation, from a protocol state machine with state invariants. The invariants are manually added to each state of the state machine. Our approach uses labeled PSMs to generate the contracts, and the labels are automatically produced from the sequence diagrams.

In the work of [26], the functionality specification of a transition is specified in terms of a design as a pair of pre-/postconditions, and an integrated specification of method m is defined as disjunction of the functionality specifications of all transitions that may react to the invocation of m. The approach integrates state machines with classes and gives complete behavior definitions for the methods. The work of [13] provides full model and code generation for the approach. The transformation for the generation of method bodies presented in this paper is also based on this approach, but with an extension to support concurrency and critical constructs.

7 Conclusion

In this paper, we propose an approach and its tool implementation for automatically processing sequence diagrams of use cases into controller classes with method pre- and postconditions as the contracts that enforce the semantics of the sequence diagrams through an intermediate step of generating protocol state machines. We support the standard features found in UML 2.0 sequence diagrams and take into account concurrency and critical sections. The generated model can serve as a starting point for the next phase of model-driven development, for example, detailed object-oriented design, or be used to generate program code. With the help of the class contracts, designers can more focus on realizing the data functionality of a method itself, less worrying about in which system states the method is allowed to be invoked.

The approach is specified as several model transformations using the graphical notation of QVT Relations. The QVT-R specifications are automatically transformed to executable XSLT programs and integrated into a CASE tool. We expect the tool to be useful in bridging the gap between requirements and design models.

However, system requirements are usually specified as a set of use cases, and there exist relationships among them. A use case may be a variation of some other use case (the extend relationship) or may incorporate one or more use cases using the include relationship. Without support to the relationships, the scalability and usability of our approach will be seriously affected. A solution for the problem is considered

as future research. In addition, UML includes state invariants as a way of defining additional conditions along the lifelines of sequence diagrams. These conditions can be added to the pre-/postconditions of the corresponding transitions in the generated state machines. This mechanism allows very convenient and fine-grained control over sequence diagrams, and we consider to support it in our approach. Moreover, as the generated state machine could become quite complicated, it is necessary to optimize it by removing redundant states and null transitions. Also a readable layout of the state machine is a challenging problem, as we produce the graphical representations for the state diagrams as well. Future work includes applying the tool to more middle- and large-scale case studies, for the purpose of promoting the usefulness and ease of use of the tool, and evaluating its practical applicability in real software projects.

The transformations described in this paper are not isolated. They are part of model transformations we developed to support all phases of component-based, model-driven development [6]. These will cover small-step structural refinements of the class models, OO design through applying design patterns, and the mapping of OO- to component-based models [27, 28]. However, there are still many challenges in automating these model transformations, and our work is ongoing.

Acknowledgements

This work was supported by the PEARL, GAVES (Macau Science and Technology Development Fund), Guizhou International Scientific Cooperation Project (Grant No. G[2011]7023), and the National Natural Science Foundation of China (Grant Nos. 61073022, 61103013, 91118007).

References

1 France R, Rumpe B. Model-driven development of complex software: A research roadmap. In: Lionel C B, Alexander L W, eds. 2007 Future of Software Engineering (FOSE 07). Minnesota: IEEE, 2007. 37–54
2 Schmidt D. Model-driven engineering. IEEE Comput, 2006, 39: 25–31
3 Hailpern B, Tarr P. Model-driven development: The good, the bad, and the ugly. IBM Syst J, 2006, 45: 451–461
4 OMG. Unified Modeling Language: Superstructure, version 2.4.1. Aug. 2011. http://www.omg.org/spec/UML/2.4/Superstructure
5 Larman C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 3rd ed. New Jersey: Prentice-Hall, 2005
6 Ke W, Li X S, Liu Z M, et al. rCOS: A formal model-driven engineering method for component-based software. Front Comput Sci China, 2012, 6: 17–39
7 Dromey R G. From requirements to design: Formalizing the key steps. In: Antonio C, Peter L, eds. Proceedings of First International Conference on Software Engineering and Formal Methods (SEFM 03). Brisbane: IEEE, 2003. 2–11
8 Leavens G, Baker A, Ruby C. JML: A notation for detailed design. Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers Group, 1999. 175–188
9 OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.1. Jan. 2011
10 Li D, Li X S, Stolz V. QVT-based model transformation using XSLT. SIGSOFT Softw Eng Notes, 2011, 36: 1–8
11 Ali J, Tanaka J. Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams. J Comput Sci Inf Manage, 2001, 2: 22–34
12 WWW Consortium, XSL Transformations (XSLT) Version 2.0, W3C Recommendation. Jan. 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/
13 Chen Z B, Morisset C, Stolz V. Specification and validation of behavioural protocols in the rCOS modeler. In: Proceedings of 3rd Intl. Symp. on Fundamentals of Software Engineering (FSEN 2009), Springer, 2010. LNCS 5961: 387–401
14 Hoare C A R. Communicating Sequential Processes. New Jersey: Prentice-Hall, 1985
15 Formal Systems (Europe) Ltd. FDR2 User Manual, 2005. http://www.fsel.com
16 Whittle J, Jayaraman P. Generating hierarchical state machines from use case charts. In: Martin G, Robyn L, eds. Proceedings of the 14th IEEE International Conference on Requirements Engineering. Minneapolis: IEEE, 2006. 19–28
17 Amyot D, Eberlein A. An evaluation of scenario notations and construction approaches for telecommunication systems development. Telecommun Syst, 2003, 24: 61–94
18 Liang H, Dingel J, Diskin Z. A comparative survey of scenario-based to state-based model synthesis approaches. In: Proceedings of 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools. ACM,

2006. 5–12

19 Whittle J, Jayaraman P. Synthesizing hierarchical state machines from expressive scenario descriptions. ACM T Softw Eng Meth, 2010, 19: 8

20 Grønmo R, Møller-Pedersen B. From UML 2 sequence diagrams to state machines by graph transformation. J Object Technol, 2011, 10: 1–22

21 Ziadi T, Helouet L, Jezequel J. Revisiting statechart synthesis with an slgebraic approach. In: Proceedings of the 26th International Conference on Software Engineering. Edinburgh: IEEE, 2004. 242–251

22 Graaf B, Deursen A V. Model-driven consistency checking of behavioural specifications. In: Proceedings of the Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 07). Braga: IEEE, 2007. 115–126

23 Harel D, Kugler H, Pnueli A. Synthesis revisited: Generating statechart models from scenario-based requirements. In: Formal Methods in Software and Systems Modeling. Berlin: Springer, 2005. LNCS 3393: 309–324

24 Harel D, Segall I. Synthesis from scenario-based specifications. J Comput Syst Sci, 2012, 78: 970–980

25 Porres I, Rauf I. Generating class contracts from UML protocol statemachines. In: Proceedings of 6th International Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa'09). ACM, 2009. 8

26 Chen X, Liu Z M, Mencl V. Separation of concerns and consistent integration in requirements modelling. In: Proceedings of 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2007). Berlin: Springer, 2007. LNCS 4362: 819–831

27 Li D, Li X S, Liu Z M, et al. Interactive transformations from object-oriented models to component-based models. In: Proceedings of Formal Aspects of Component Software (FACS 11). Berlin: Springer, 2012. LNCS 7253: 97–114

28 Li D, Li X S, Liu Z M, et al. Support formal component-based development with UML profile. In: Proceedings of the 22nd Australian Conference on Software Engineering (ASWEC 2013). Melbourne: IEEE, 2013. 191–200