

From UML 2 Sequence Diagrams to State Machines by Graph Transformation

Roy Grønmo^a Birger Møller-Pedersen^b

a. SINTEF Information and Communication Technology, Oslo, Norway

b. Department of Informatics, University of Oslo, Norway

Abstract Algebraic graph transformation has been promoted by several authors as a means to specify model transformations. This paper explores how we can specify graph transformation-based rules for a classical problem of transforming from sequence diagrams to state machines. The specification of the transformation rules is based on the concrete syntax of sequence diagrams and state machines. We introduce tailored transformation support for sequence diagrams and a novel graphical operator to match and transform combined fragments.

Keywords Graph transformation; Model transformation; UML; sequence diagram; state machine

1 Introduction

Although sequence diagrams and state machines are used in different phases and are made with different diagram types, there is a great deal of overlap between the two specifications. The behavior defined by the sequence diagrams should also be recognized as behavior by the state machines.

There has been a lot of efforts to transform from sequence diagram-like specification languages to state-based languages (e.g. [KGSB99, WS00, ZHJ04, Sun07]). None of the previous approaches takes full advantage of the combined fragments that were introduced in UML 2.

The combined fragments in UML 2 includes possibilities to model conditional behavior (`alt` operator) and loops (`loop` operator), and these can have guard expressions and be arbitrarily nested. A combined fragment is displayed with a rectangle that spans the involved lifelines, an operator type shown in the top left corner of the rectangle, and dashed horizontal lines as operand separators in cases with multiple operands.

In this paper we specify a transformation from sequence diagrams to state machines where the specified rules are based on the concrete syntax of sequence diagrams

and state machines. Our approach differs from the traditional model and graph transformation approaches, where transformations are specified in relation to the abstract syntax. We claim that concrete syntax-based rules are more user-friendly since the specifier does not need to have knowledge of the metamodels and the associated abstract syntax. This is particularly useful for sequence diagrams where the abstract syntax is complicated and quite different from the concrete syntax.

We introduce a *fragment operator* that allows us to specify the matching and transformation of combined fragments with an unknown number of operands. Our rules are mapped to traditional graph transformation rules and the transformation takes place in the AGG tool [Tae03].

The remainder of this paper is structured as follows. In Section 2 we briefly introduce sequence diagrams, state machines, and the notion of a trace-based refinement theory; Section 3 describes a possible modeling process that starts with sequence diagrams and evolves to state machines; Section 4 describes preliminaries on graph transformation; Section 5 explains how we can define transformation rules based on the concrete syntax of sequence diagrams and state machines; Section 6 presents the specialized transformation formalism for sequence diagrams and our set of transformation rules from sequence diagrams to state machines; Section 7 compares our approach with related work; and finally Section 8 concludes the paper.

2 Sequence Diagrams, State Machines and Refinement

Figure 1 shows a sequence diagram and a corresponding state machine to represent the behavior of the second lifeline object (`GasPump`) in the sequence diagram. The sequence diagram has two lifelines with the types `User` and `GasPump`, and two messages with the signals `insertCard` and `requestPin`. A lifeline, visualized with a rectangle and a dashed line below, represents an interacting entity on which events take place in an order from top to bottom on the dashed line. Each message is represented by two events, a send event (at the source of the message arrow) and a receive event (at the target of the message arrow).

In this paper we only use sequence diagrams with asynchronous messages, although our transformation apparatus works for both synchronous and asynchronous messages. Asynchronous messages fits nicely with the event-based nature of state machines, unlike sequence diagrams with synchronous messages that have a procedural nature. We omit the optional rectangles to visualize when a lifeline is active, since these are more relevant for synchronous messages.

A state machine, consistent with the `GasPump` lifeline, has an initial state with a transition leading to the state named `Idle`. The `Idle` state has one outgoing transition, with `insertCard` as its trigger and `requestPin` as its effect, going to the final state.

The semantics of a sequence diagram can be described as a set of *positive traces*

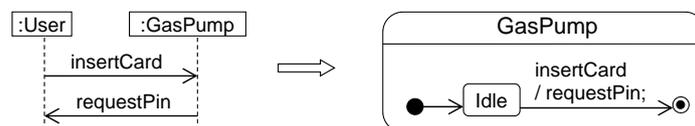


Figure 1 – Consistency between sequence diagram and state machine

and a set of *negative traces* [RHS05]. *Positive traces* define valid behavior and *negative traces* define invalid behavior, while all other traces are defined as inconclusive. In the sequence diagram of Figure 1, there is exactly one positive trace: $\langle \text{send insertCard}, \text{receive insertCard}, \text{send requestPin}, \text{receive requestPin} \rangle$

Negative traces are described by special operators (e.g. **neg**), which are not used in the diagram of Figure 1. Hence, all other traces than the single positive trace, are inconclusive.

The leftmost part of Figure 2 shows a graphical notation of the universe of traces, where a circle is divided into positive, inconclusive and negative traces. In reality there are infinitely many inconclusive traces for the sequence diagrams, and infinitely many negative traces for the state machines.

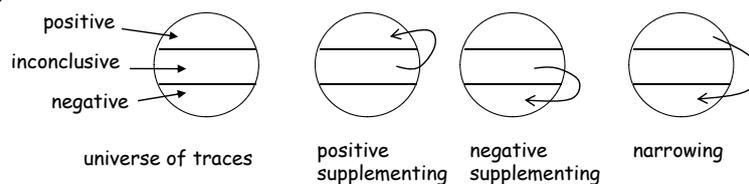


Figure 2 – Universe of traces and refinement

The rest of Figure 2 shows the three kinds of sequence diagram refinement that are defined by STAIRS [RHS06]. (1) *positive supplementing*. A previously undescribed scenario is described as positive behavior. (2) *negative supplementing*. A previously undescribed scenario is described as negative behavior. (3) *narrowing*. Some previously described positive behavior is described as negative behavior.

The set of sequence diagrams describing a system will normally have a non-empty set of inconclusive traces, which we call a *partial specification*. An actual implementation may choose to implement the inconclusive traces as either positive or negative. A state machine on the other hand, has no inconclusive traces and is thus a *complete specification*.

Since the set of sequence diagrams is only a partial specification, the automatically produced state machines are only intended to be a good starting point for a manual refinement. This makes it important that the produced state machines are readable.

3 A Modeling Process from Sequence Diagrams to State Machines

In Figure 3 we show our recommended modeling process of four steps, starting with the early phase of simple sequence diagrams and ending with the final state machines that can be used to generate Java code [HMP00]. One column shows the refinement types that are typical for each step. The artefact result of each step is described in the third and fourth column. For each artefact we also show the universe of traces in the last column to illustrate how the sizes of the three trace sets (positive, inconclusive, negative) typically evolve throughout the modeling process.

Step 1 - Initial modeling. Scenarios can easily be described with intuitive and simple diagrams showing example executions in the to-be-implemented system. These initial sequence diagrams should not be too detailed and they should use few or no combined fragments, since this could be counterproductive in the idea and brainstorming phase. Similar behavior may occur in several diagrams. This step can

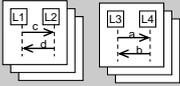
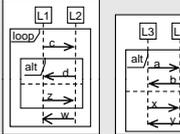
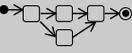
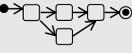
| Step | Refinement | Artefact Result | Artefact Description | Universe of Traces |
|------|---|---|--|---|
| 1 |  supplementing |  | <ul style="list-style-type: none"> • simple diagrams • duplicated behavior across diagrams |  |
| 2 |  merging, detailing, narrowing |  | <ul style="list-style-type: none"> • detailed diagrams • duplicated behavior is merged into a single diagram • combined fragments |  |
| 3 |  negative supplementing  sd2sm transformation |  | <ul style="list-style-type: none"> • initial state machine |  |
| 4 |  detailing, positive supplementing wrt. SD contract |  | <ul style="list-style-type: none"> • executable state machine |  |

Figure 3 – Modeling process from sequence diagrams to state machines

be seen as a positive and negative supplementing, since prior to the modeling all traces are inconclusive.

Step 2 - Detailed modeling. Combined fragments are used to manually merge similar behavior from multiple diagrams into a single diagram. The advantage is that this all happens in the context of the well-known sequence diagrams with no need to clutter the sequence diagrams with other expressions, nor a need to master another description language. An existing tool can be used to check that the modified sequence diagrams are refinements of the previous sequence diagrams [Lun07].

The merging into a single diagram can always be achieved by using enough combined fragments. For convenience, unrelated scenarios involving the same lifeline can be kept in several diagrams, followed by a transformation that merges all lifelines into the same diagram. This transformation can introduce one outermost `alt` operator with one operand for each of the unrelated scenarios.

In this step, we typically add some guards to alternative behavior operands and detail previous diagrams, which means that we perform a narrowing. The step 2 artefact represents a *contract* (named 'SD contract' in the figure) which an implementation must fulfill. We interpret all the positive traces as mandatory behavior which must be implemented, while the negative traces describe prohibited behavior.

Step 3 - Generate State Machine. Our automated generation `sd2sm`, in step 3, makes a state machine that accepts all positive traces from the sequence diagrams. Inconclusive traces are not implemented, and these traces become negative. Hence, step 3 performs a negative supplementing.

Step 4 - Refine State Machine. In step 4, the modeler refines the generated state machines so that they are detailed enough to express a full implementation. Furthermore, the modeler may also freely increase the number of implemented traces, but restricted to those that are inconclusive in the SD contract (positive supplementing). Any modification of the state machines should be checked to see if the modification

represents a breach of contract. Brændshøi has implemented an automated tool that checks if a state machine is a 'proper implementation' of a set of sequence diagrams [Bræ08].

The rest of the paper focuses on our transformation rules to support step 3.

4 Preliminary: Algebraic Graph Transformation

We provide the known formal foundation of algebraic graph transformation [LEO06].

Definition 1 (Graph and Graph Morphism) A graph $G = (G_N, G_E, src, trg)$ consists of a set G_N of nodes, a set G_E of edges, two mappings $src, trg : G_E \rightarrow G_N$, assigning to each edge $e \in G_E$ a source node $src(e) \in G_N$ and target node $trg(e) \in G_N$.

A graph morphism $f : G_1 \rightarrow G_2$ from one graph to another, with $G_i = (G_{E,i}, G_{N,i}, src_i, trg_i)$, ($i = 1, 2$), is a pair $f = (f_E : G_{E,1} \rightarrow G_{E,2}, f_N : G_{N,1} \rightarrow G_{N,2})$ of mappings, such that $f_N \circ src_1 = src_2 \circ f_E$ and $f_N \circ trg_1 = trg_2 \circ f_E$ (preserve source and target). A graph morphism $f : G_1 \rightarrow G_2$ is injective if f_N and f_E are injective mappings.

Only injective graph morphisms will be relevant in this paper.

Definition 2 (Rule) A graph transformation rule $p : L \xleftarrow{l} I \xrightarrow{r} R$ consists of three graphs L (LHS), I (Interface) and R (RHS) and a pair of injective graph morphisms $l : I \rightarrow L$ and $r : I \rightarrow R$.

Definition 3 (Match and Dangling Condition) Given a graph G and a rule $p : L \xleftarrow{l} I \xrightarrow{r} R$. Then an occurrence of L in G , i.e. an injective graph morphism $m : L \rightarrow G$, is called match.

The function $isMatch : L \times G \times (L \rightarrow G) \rightarrow Bool$ returns true if and only if $L \rightarrow G$ is a match of L in G . A match m for rule p satisfies the dangling condition if no node in $m(L \setminus l(I))$ is incident to an edge in $G \setminus m(L \setminus l(I))$.

Definition 4 (Derivation Step) Given a graph G , a graph transformation rule $p : L \xleftarrow{l} I \xrightarrow{r} R$, and a match $m : L \rightarrow G$, then there exists a derivation step from the graph G to the graph H if and only if the dangling condition is satisfied. H is constructed as follows:

1. Remove the image of the non-interface elements of L in G , i.e. $H' = G \setminus m(L \setminus l(I))$.
2. Add the non-interface elements of R into H , i.e. $H = H' \cup (R \setminus r(I))$.

A negative application condition [LEO06] is an extension of the LHS which prevents matches from being applied in a derivation step.

Definition 5 (Negative Application Condition (NAC)) A NAC for a graph transformation rule $L \xleftarrow{l} I \xrightarrow{r} R$, is defined by a pair of injective graph morphisms: $L \xleftarrow{s} NI \xrightarrow{t} N$, where N is the negative graph, and NI defines the interface graph between L and N .

A match $m : L \rightarrow G$ satisfies the NAC if and only if there does not exist an injective graph morphism $n : N \rightarrow G$ which preserves the NI interface mappings, i.e.

for all nodes v in NI we have $n_N(t_N(v)) = m_N(s_N(v))$ and for all edges e in NI we have $n_E(t_E(v)) = m_E(s_E(e))$.

A rule can have an arbitrary number of NACs, and a derivation step can only be applied if a match satisfies all the NACs of the matched rule.

In addition to the above, we adopt the theory of *typed attributed graphs* [HKT02], where graphs are extended by assigning types to nodes and edges, and by assigning a set of named attributes to each node type. A graph morphism must now also preserve the node and edge types, and the attribute values.

In the graph transformation rules throughout this paper we only explicitly display the LHS and the RHS graphs, while the interface graph is given by shared identifiers of elements in the LHS and the RHS/NACs.

A *collection operator* [GKMP09] can be used in a rule to match and transform a set of similar subgraphs in one step. This is also possible with so-called rule amalgamation [Tae96]. We will use the collection operator since it provides a notation that can be integrated into a single rule. With rule amalgamation, there will be one subrule to capture the rule part outside of all subgraphs, and one elementary rule for each subgraph to be matched and transformed.

A dotted frame is used to visualize a collection operator, where all the contained nodes and edges are placed inside the frame. A shared identifier of a collection operator in the LHS and the RHS/NACs denotes a collection operator in the interface graph. The identifier and cardinality of a collection operator is visualized next to the collection operators dotted frame. There can be multiple collection operators, but two collection operators must be specified such that they cannot match the same nodes or edges.

The set of all collection operators in a rule $p : L \xleftarrow{l} I \xrightarrow{r} R$ is referred to as $Coll_p$. We use ψ to denote a function that maps each collection operator, in a rule p , to a number within its cardinality range, i.e. $\psi : Coll_p \rightarrow (\mathbf{N} = \{0, 1, 2, \dots\})$, where $\forall c \in Coll_p : \psi(c) \in [c.min, c.max]$.

We let $p^\psi : L^\psi \xleftarrow{l^\psi} I^\psi \xrightarrow{r^\psi} R^\psi$ denote the collection free rule where each collection operator c in p is replaced by $\psi(c)$ number of collection content copies. In these copies all the copied elements/attributes get fresh identifiers/variables respectively, while the interface elements between the pointcut and the advice are maintained.

The minimal configuration of ψ , denoted ψ^- , for which we can find a match for a rule is when $\forall c \in Coll_p : \psi(c) = c.min$. In the matching process we look for a match of the collection free rule p^{ψ^-} . Then, each collection operator match and the ψ is extended as much as possible to achieve a complete match. This results in a dynamically built rule p^ψ with a match upon which we can try to apply a derivation step according to Definition 4.

5 Our Transformation Rules are Specified in the Concrete Syntax

The *concrete syntax* of a diagram type uses a tailored visualization with icons and rendering rules depending on the element types. To improve the usability for the graph transformation designer, we define the transformation rules upon concrete syntax. A clear benefit for the user is that the specification of the rules does not require knowledge of the often complicated metamodels of the involved source and target languages.

Our approach is depicted in Figure 4. The source model sequence diagram, the sequence diagram to state machine rules (SD2SM) and the resulting state machine are all represented in the concrete syntax. The matching and transformation of the sequence diagram part of the concrete syntax-based rules is formally defined in Section 6.3. The mapping from concrete syntax to abstract syntax-based rules ensures that the formal definitions from Section 6.3 are preserved by the resulting graph transformation rules which are formally defined in Section 4.

As with algebraic graph transformation, our rules use a LHS, a RHS, and an implicit interface model defined by identifiers which are displayed next to its corresponding element. The LHS and the RHS can both be a mix of sequence diagrams and state machines, and our transformation rules use an ordinary graph edge to link a lifeline to a state.

Our rules are automatically transformed into traditional abstract syntax rules, where we have a tailored support for (1) the parent state relation, (2) the ordering of occurrences on a lifeline, and (3) combined fragments.

All states and transitions in a state machine model, except the outermost state, have a parent state. Together with the dangling condition, this means that we cannot delete a state or a transition without also matching the parent state. Furthermore, new states and transitions must also get the proper parent state. For convenience, we include an implicit parent state around the whole LHS and the RHS state machine models.

Except for the implicit parent state, the state machine part of our rule models can basically use the same abstract syntax representation as ordinary state machine models. The matching and transformation of the state machine part of the rules can be directly understood by translating the state machines into abstract syntax. This makes the state machine support quite trivial in our approach. For this reason we do not have an equivalent version of Section 6.3 for state machines.

For the sequence diagram part of our rules, however, we introduce a fragment operator and tailored matching and transformation definitions. This special treatment of sequence diagrams is incorporated into the mapping to abstract syntax rules, such that ultimately plain algebraic graph transformation is used.

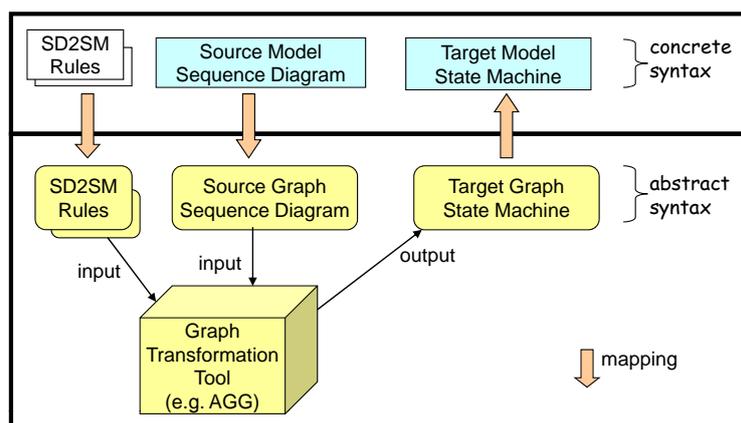


Figure 4 – Relationship between concrete syntax and abstract syntax

6 Transformation of Sequence Diagrams

Figure 5 shows our simplified metamodel for UML 2 sequence diagrams. A sequence diagram is represented by a set of lifelines. A lifeline has a top-down ordered sequence of occurrences.

An occurrence can be one of five kinds (`event`, `combinedFragment`, `start`, `end`, `arbEvt`), where only events or combined fragments conceptually occur on an ordinary sequence diagram lifeline. The meta occurrence of kind `start` shall be the very first occurrence on a lifeline, and the meta occurrence of kind `end` shall be the very last occurrence on a lifeline. These meta occurrences enables us to easily specify the replacement of a subsequence of occurrences on a lifeline.

Finally, an occurrence of kind `arbEvt` represents the lifeline symbol called *arbitrary events*, which was previously introduced in [GSMPK08]. This symbol allows matches to have an arbitrary number of occurrences in the symbol's position. Generally, the symbol can be placed anywhere on a lifeline. In this paper we restrict the usage to at most one symbol per lifeline and if used it shall be placed as the very first occurrence on the lifeline. This restriction is sufficient for our transformation from sequence diagrams to state machines, and allows us to focus on the contributions of this paper.

A message consists of a send event and a receive event, which are normally placed on two different lifelines. A combined fragment spans over many lifelines and it has one or more operands. A combined fragment with operator `opt`, `loop` or `neg` contains exactly one operand, while for other operators (e.g. `alt`, `par`) it contains an arbitrary number of operands.

Each operand has a guard attribute and spans over a subset of the lifelines which its combined fragment spans over. An operand lifeline has a `partOf` relation to indicate to which lifeline it belongs.

As our example we will use the sequence diagram (Figure 6), named `GasPump`, that describes a gas pump scenario. A user inserts a payment card (`insertCard`). The gas pump requests the pin code from the user (`requestPin`) and the user enters the pin code (`pinCode`). A bank validates the pin code (`validate` and `result`), and an `alt` operator models the two possible outcomes: 1) *valid pin code*: The user is informed to start fuel (`startFuel`) and the user indicates end of fueling by hanging up the gas pump (`hangUp`), or 2) *invalid pin code*: The user is informed that the entered pin code is invalid (`invalidPin`). In both cases, the scenario ends by ejecting the card (`cardOut`).

Figure 6a shows the sequence diagram in the well-known concrete syntax, while Figure 6b shows a possible abstract syntax of the same diagram according to the metamodel we have defined above. A few metamodel properties are shortened for

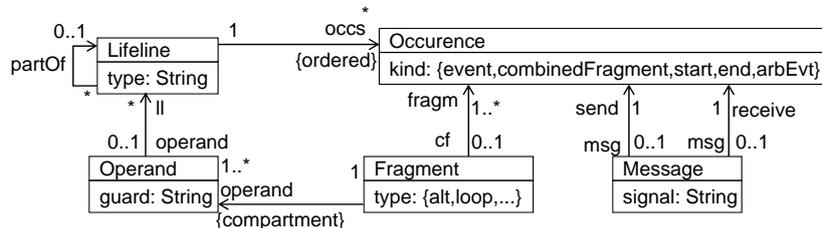
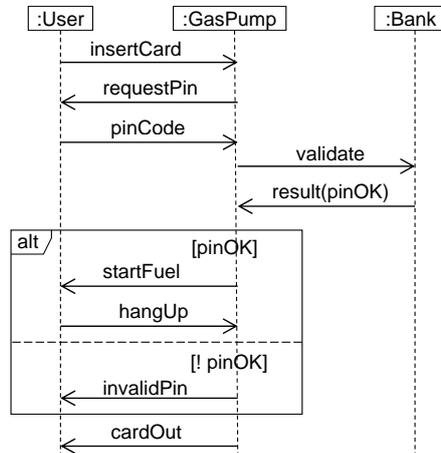
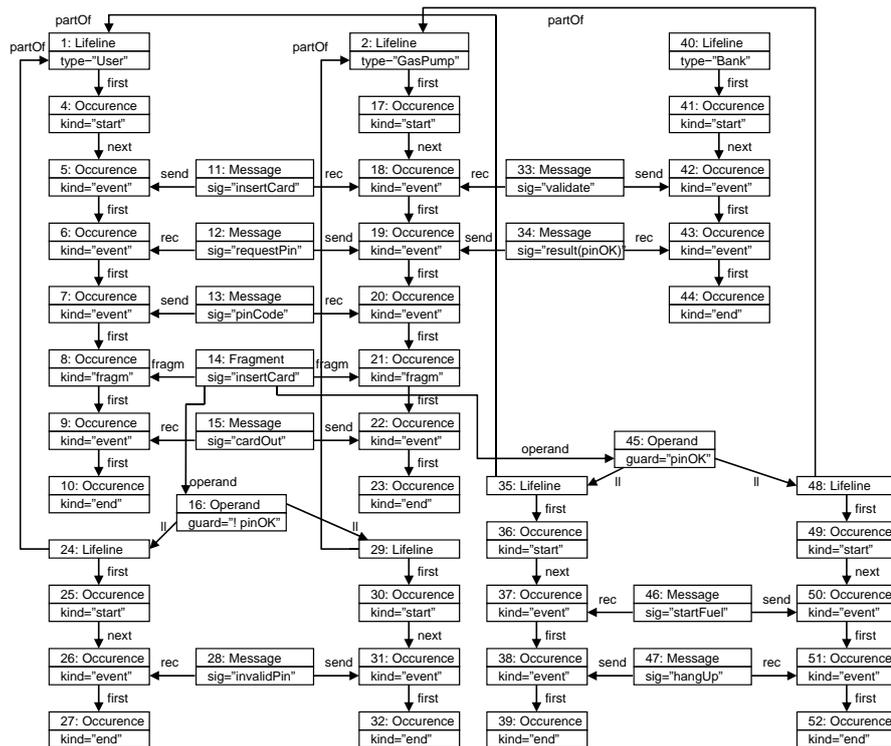


Figure 5 – A simplified metamodel for UML 2 sequence diagrams

brevity: **fragm** = combinedFragment, **sig** = signal, and **rec** = receive. Even though the sequence diagram is fairly simple, the abstract syntax diagram is complicated. For a modeler it is obviously preferable to model sequence diagrams by using a standard sequence diagram editor that allows working in concrete syntax rather than in abstract



(a) Concrete syntax



(b) Abstract syntax

Figure 6 – The GasPump model

syntax. In this paper we will show that concrete syntax can have the same benefit with respect to the specification of graph transformation rules.

6.1 Fragment Operator

In the transformation rules (e.g. the `Alt` rule shown later) there is a need to match a combined fragment with an unknown number of operands, and to keep only the operand parts in the RHS of a rule. In the standard concrete syntax of sequence diagrams it is not straightforward to distinguish between the combined fragment operator itself and its operands. A similar challenge applies to state regions of state machines, which are also displayed in separate compartments of a state. We call such relations for a compartment relation and indicate this by the tag `{compartment}` in the metamodel (Figure 5).

For relations that are tagged as compartment in the metamodel, we provide a new graphical element in the rules. For sequence diagrams we call this element a *fragment operator*. It is displayed as an ordinary combined fragment rectangle with a set of rectangles labeled 'operand' inside to denote the fragment operands. The fragment operator has a clear border between itself and its operands, as opposed to the syntax of ordinary sequence diagrams.

Multiple operands are expressed by explicitly drawing several compartment operands, or by placing a collection operator around a compartment operand as illustrated by the rule in Figure 7a. Notice that the rule in concrete syntax is very concise compared to the relatively complicated corresponding rule in abstract syntax (Figure 7b).

The semantics of the rule can be explained as follows. A match shall have a combined fragment of type `alt` as the first occurrence on some lifeline identified by `id=1`. The abstract syntax rule ensures this by requiring that the combined fragment is the first occurrence after the meta-occurrence `start` on a lifeline with identifier 1. The NAC introduced in the abstract syntax requires that a lifeline specified in the concrete syntax is not part of an operand. Such a fixed NAC is introduced for all LHS lifelines so that we can only match a lifeline which is not part of another lifeline.

The collection operators with ids `c2` and `c3` are introduced by the mapping to abstract syntax rule, and they allow a matching combined fragment to span across lifelines not specified by the concrete syntax rule. Furthermore, these collection operators enables us to delete the combined fragment even though some of its lifelines are not explicitly matched by the concrete syntax rule.

When the combined fragment operator is removed, and its operands are kept, the `ll` edge to the part lifelines with identifier `id=5` is removed, and these lifelines are no longer prevented from matches by the generated fixed NACs in the abstract syntax rules.

Notice our notation `partLL(id=1)` in Figure 7a, which is used to identify a RHS lifeline. This will retrieve the lifeline that corresponds to the lifeline with `id=1` within the particular operand. Figure 7b shows the abstract syntax version of the rule, where the `partLL` lifeline has a `partOf` relation to the `id=1` lifeline. The effect of this rule is the same as when we leave out the `partLL` lifeline. However, the `partLL` notation is useful for rules that need to update a `partLL` lifeline, such as we will see later for all our rules that use the fragment operator (`Alt`, `Loop`, `Par`, and `Opt`). There the lifeline gets a relation to a corresponding state in a state machine.

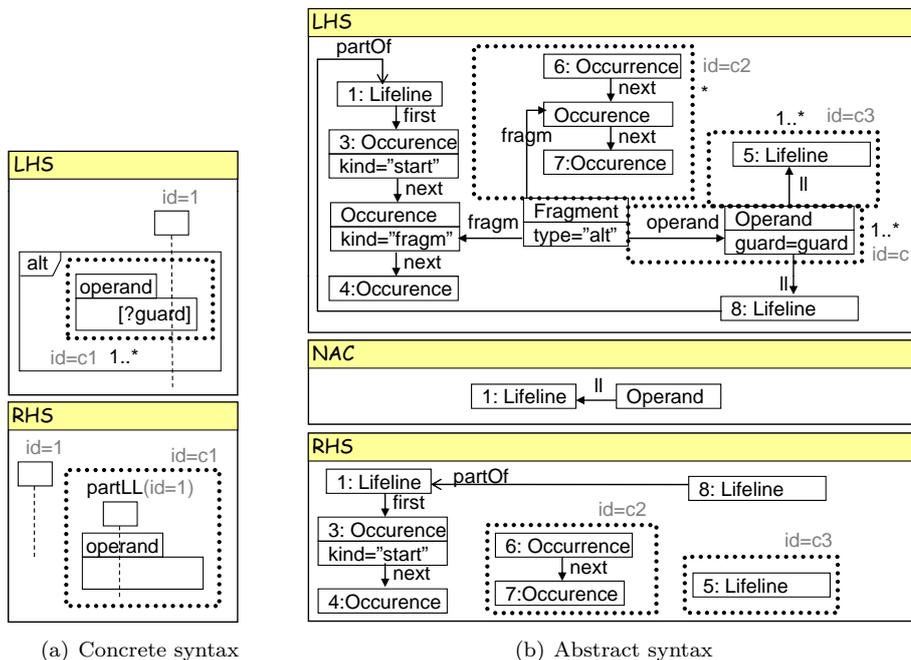


Figure 7 – Mapping a rule with the fragment operator from concrete to abstract syntax

6.2 Transformation Rules

In this section we present the transformation rules, and we show how the rules gradually transform from a sequence diagram into state machines. We will use the model in Figure 6a as the input model of the transformation.

Each lifeline corresponds to a state machine. When producing a state machine, it is sufficient to look at the single corresponding lifeline with its events and how these events are structured within the combined fragments. A prerequisite to this claim is that each lifeline occurs only in one sequence diagram, which is ensured by introducing the combined fragments in step 2 of the method described in Section 2.

The intermediate models in the transformation process contains sequence diagrams, state machines and helper edges (abstract syntax edges) with type name **state** to link a lifeline to its current position in the corresponding state machine.

The transformation process takes a lifeline type as input so that we can produce a state machine for that lifeline. A rule called **InitSM** (Figure 8a) simply adds a new state machine with the same name as the given lifeline type and adds an initial state with a transition leading to a state called **Idle**. The rule adds the edge of type **state** from the lifeline to the **Idle** state. A NAC ensures that the **InitSM** rule is applied exactly once. The intermediate GasPump model after applying the **InitSM** rule is seen in Figure 8b.

The transformation rules then proceed by matching the top-most occurrence on the lifeline, adding corresponding behavior to the state machine and removing the treated occurrence from the lifeline. Removing an occurrence normally means that we need to delete an occurrence also from another lifeline, e.g. removing the send event from a lifeline can only be done if we also remove the receive event of the message.

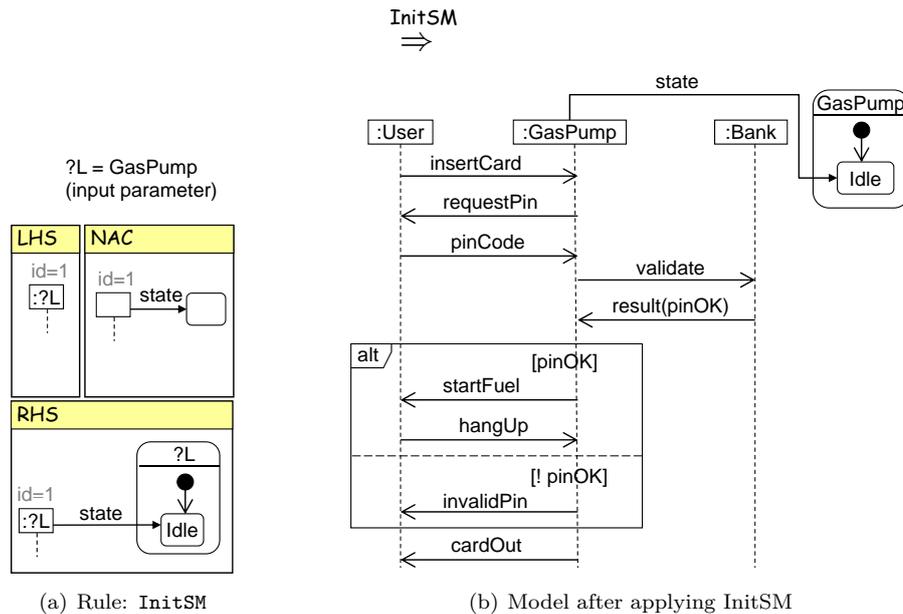


Figure 8 – GasPump: The InitSM rule creates the state machine.

A top-most 'occurrence' is either a combined fragment or an event which is part of a message. The rule **Receive** (Figure 9a) pops a receive event (and its corresponding send event from another lifeline), adds a state which now becomes the current state, and adds a transition with trigger labeled by the message name. The transition goes from the previous current state to the new current state. We use an **arbEvt** symbol to indicate that the matched send event does not need to be the very first occurrence on its lifeline. The rule **Send** (Figure 9b) pops a send event (and its corresponding receive event from another lifeline) and adds a corresponding effect on the incoming transition to the current state.

The model in Figure 9c shows the result after applying the rule sequence $\langle \text{Receive}, \text{Send}, \text{Receive}, \text{Send}, \text{Receive} \rangle$. We have omitted the **Bank** lifeline from this model and the following models in this transformation, since it has no more events.

The rule **Alt** in Figure 10a pops an **alt** fragment and makes the current state into a composite state by adding internal behavior: an initial state, an **Idle** state and a final state. For each **alt** operand we make an inner composite state.

We produce a transition from the **Idle** state to each inner composite state, where the transition guard is equal to the corresponding **alt** operand guard. The **Alt** rule uses the fragment operator to detach each **alt** operand from its **alt** operator. The operand part lifeline corresponding to the **GasPump** lifeline is referred to by the **partLL(id=1)**. An edge of type **state** is added from the part lifeline to the **Idle** state of the inner composite state. Notice how the collection operator allows us to express the treatment of multiple operands. In the RHS, the **partLL(id=1)** lifeline, the operand and the inner composite state are all inside the collection operator. This means that we get one occurrence of all these elements for each **alt** operand.

Finally the original lifeline (referred to by **id=1**) where we popped the **alt** operator, gets a new state as its current state. The old current state gets a transition leading to

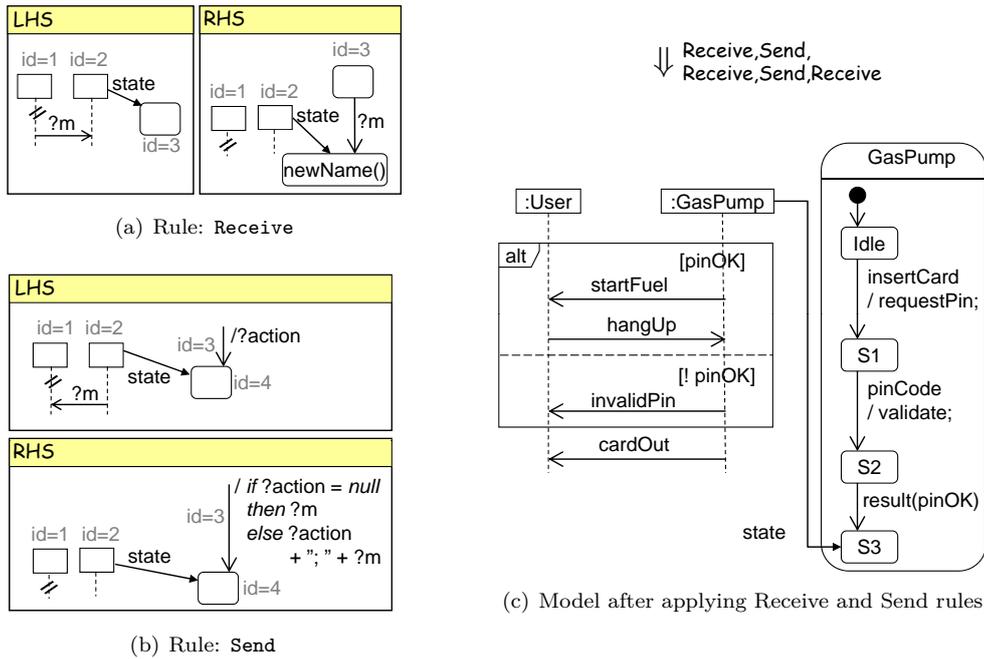


Figure 9 – GasPump: Applying Send and Receive rules

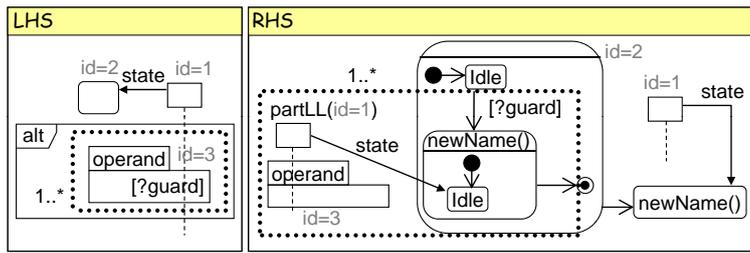
the new current state.

The model in Figure 10b shows the result after applying the **Alt** rule. Notice that we now have three sequence diagrams with **state** links to the state machine. Two of these are for the **alt** operands, and the third is for the remaining part of the sequence diagram after the original **alt** operator.

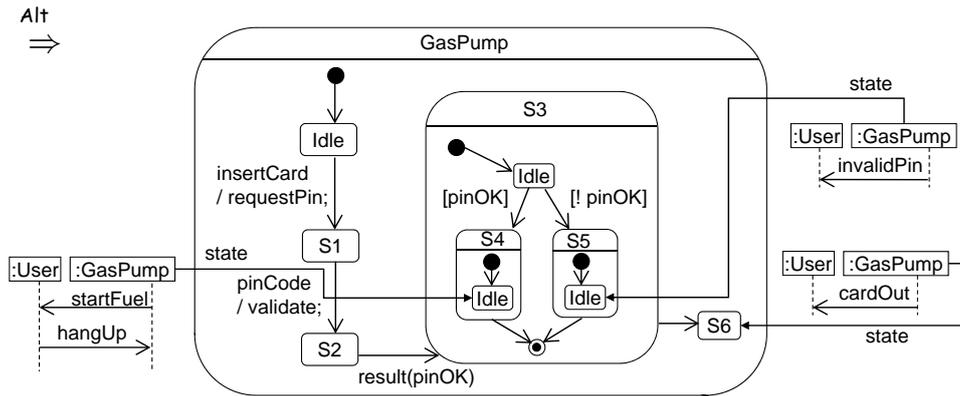
When we have mapped and removed all events from a lifeline, then we use a rule called **FinalState** (Figure 11a). The rule replaces the current state (indicated by the **state** edge) by a finalnode. Furthermore, the **state** edge and the lifeline is deleted. The dangling condition ensures that the rule only can be applied when there are no events left on the lifeline.

Figure 11b shows the intermediate model after applying three **Send** rules, one **Receive** rule and then three **FinalState** rules. The **Send** and **Receive** rules will consume all remaining messages on the three sequence diagrams. The **FinalState** rule can then be applied to remove all these three sequence diagrams.

We have now reached a state machine corresponding to the **GasPump** lifeline. It is possible to optimize the produced state machine by flattening some of the composite states. For our example we need three flattening rules which are shown in Figure 12a-c. The **FlattenIntoChoice** rule flattens the composite state holding all the internal choices corresponding to the **alt** operands. We are able to flatten the state by introducing a choice node with outgoing branches to the choices and finally a merge node with incoming branches from all the choices. The **FlattenSubState1** rule flattens a composite state of only one transition, while the **FlattenSubState2** rule flattens a composite state which also holds internal states. By applying the three flattening rules we produce a more readable and concise state machine in Figure 12d (called the target model) than we had in Figure 11b.

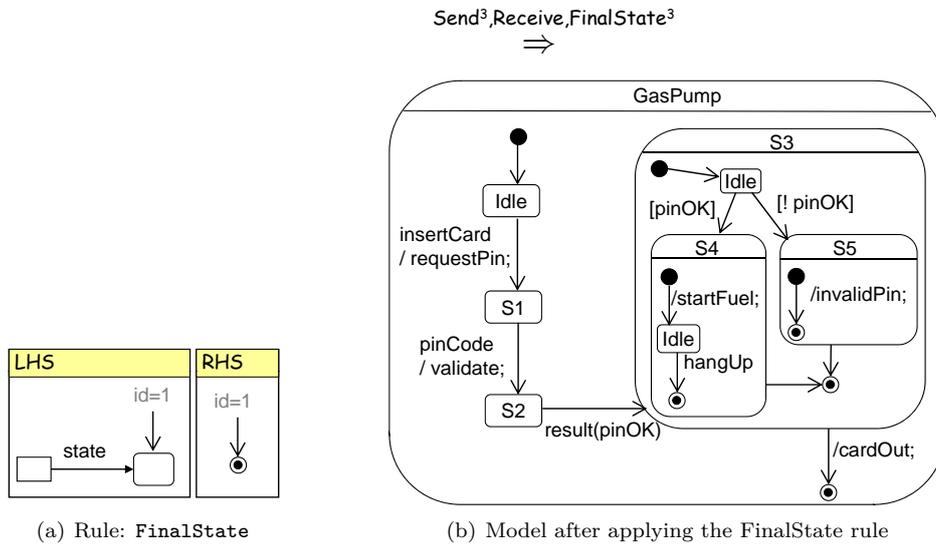


(a) Rule: Alt



(b) Model after applying the Alt rule

Figure 10 – GasPump: Applying the Alt rule.



(a) Rule: FinalState

(b) Model after applying the FinalState rule

Figure 11 – GasPump: Applying the FinalState rule.

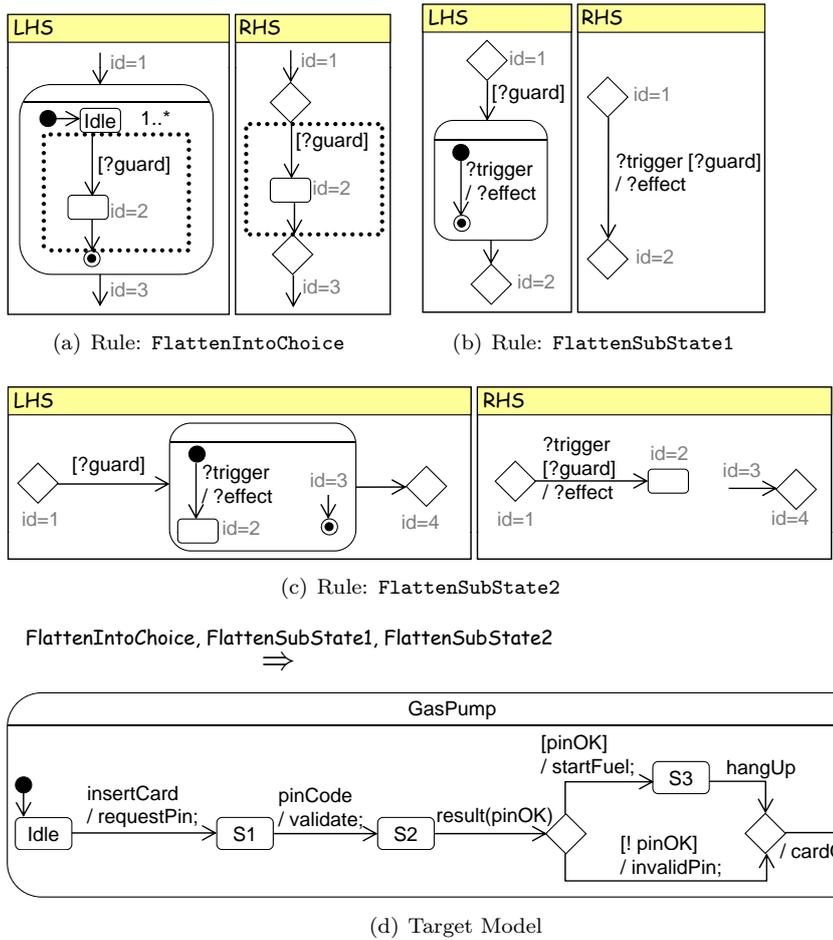


Figure 12 – GasPump: We reach the target model after applying the flattening rules.

The transformation produces one state machine per lifeline, and each state machine is placed in a region within a combined outermost state machine. This means that all the state machines are started in parallel.

We have also defined mapping rules for other sequence diagram operators that are not used in the `GasPump` example. The `Loop` rule (Figure 13a) makes the current state into a composite state with an `idle` state inside. The `GasPump` lifeline of the `loop` operand gets a `state`-labeled edge to the `idle` state. Furthermore, the composite state has a reflexive transition with the guard condition taken from the looping condition of the `loop` operator. A transition with a negated loop guard leaves the composite state into a newly created state, which becomes the current state of the remaining sequence diagram (where the `loop` operator is removed).

The `Par` rule (Figure 13b) makes the current state into a composite state with one region state machine for each `par` operand. These region state machines each have an `idle` state that becomes the current state of the sequence diagram of the corresponding `par` operand. A transition leaves the composite state into a newly created state, which becomes the current state of the remaining sequence diagram

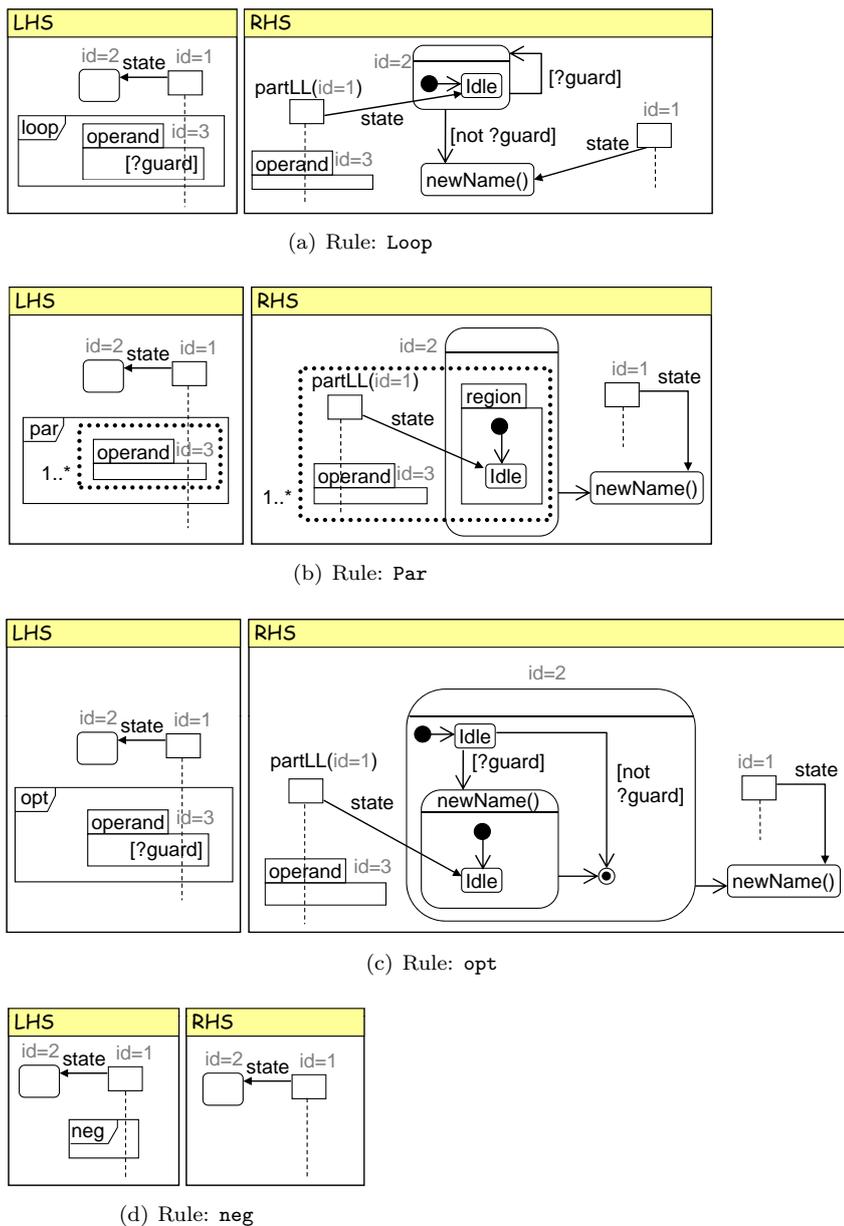


Figure 13 – More rules to generate state machine from sequence diagrams

(where the `par` operator is removed).

The `Opt` rule (Figure 13c) can be seen as the `Alt` rule defined above where there is only one operand, and where we also have the additional option of no behavior. The current state is made into a composite state where the initial state goes directly to the `idle` state. There are two outgoing transitions from this `idle` state: (1) a transition with the negative condition of the `opt` guard leading to the final state, and (2) a transition leading to a new composite state containing an `idle` state that becomes the

current state of the `opt` operand sequence diagram. A transition leaves the outermost new composite state to a new state that becomes the current state for the remaining sequence diagram (where the `opt` operator is removed).

The `Neg` rule (Figure 13d) is very simple. It simply removes the `neg` operator and its content. This is because negative behavior shall not be implemented by the state machine. The sequence diagram where all `neg` operators are ignored corresponds to positive behavior that we map to behavior of the state machine.

The transformation rules are implemented in the graph transformation tool AGG. The transformation is tested on some examples, including the `GasPump` example shown in this paper, with success. The AGG tool only supports abstract syntax rules, and we have manually translated from concrete syntax to abstract syntax rules. We have also used multiple collection free rules to simulate each rule with collection operators by following the algorithm defined in [GKMP09]. This paper defines semantics for concrete syntax-based rules of sequence diagrams which can be used to automate the translation to abstract syntax rules, as we have implemented previously for activity models [GMP08].

In graph transformation, it is a well-known principle that we need to translate the source model from concrete syntax to abstract syntax. When translating from concrete syntax-based rules to abstract syntax-based rules, much of this translation can be reused as described in Grønmo's PhD thesis [Grø09]. For diagrams like activity diagrams and state machines the translation can be almost entirely reused. As mentioned above, we only add an implicit outermost parent state to the rules LHS and RHS for the state machine abstract syntax.

The translation into abstract syntax for the sequence diagram part requires more effort than for many typical modeling languages. This is because we need to handle the ordering of events on a lifeline, distinguish between events that are the very first on a lifeline vs. in any position on the lifeline, and support for the fragment operator.

6.3 Transformation of Sequence Diagrams Formalized

This section formalizes the matching and transformation of sequence diagrams. The definitions use an injective mapping function, $\phi : L \rightarrow M$, to denote an injective mapping from the LHS elements in L to elements in the source model M . The ϕ mapping preserves the type of an element, and also all the attribute values that are specified for a LHS element.

In the definitions below a lifeline has a sequence of occurrences, where an occurrence is either an event or a combined fragment. Hence, we ignore the meta occurrences `start`, `end` and `arbEvt`, except for checking if the `arbEvt` symbol is present on a lifeline. First, we define a list of useful notation and helper definitions:

- $s \frown t$ denotes the concatenation of two (finite) sequences s and t
- $Occur^*$ denotes the set of all possible occurrence sequences
- $l.hasArbEvt$ denotes if a lifeline l has the `arbEvt` symbol on top
- $l.operand$ denotes the operand in which a lifeline l is a part and returns `null` if l is an ordinary lifeline that is not part of an operand
- $l.occs$ denotes the top-down sequence of occurrences of lifeline l

- *o.cf* denotes the combined fragment of an occurrence *o*. If the occurrence is an event, then the value is `null`
- M_{LL} denotes the set of lifelines of a sequence diagram M
- M_F denotes the set of combined fragments of a sequence diagram M

The definition below defines a match for a lifeline without an `arbEvt` symbol.

Definition 6 (Lifeline match from top) *The mapping ϕ is a lifeline match from top if and only if ϕ maps the top-down occurrence sequence O_l of a LHS lifeline to a continuous beginning subsequence of the corresponding source lifeline's top-down occurrence sequence O_s . Formally,*

$$LLMatch_{\phi}^1(O_l, O_s) \stackrel{\text{def}}{=} \exists O \in Occur^* : O_s = \phi(O_l) \frown O$$

The definition below defines a match for a lifeline with an `arbEvt` symbol.

Definition 7 (Lifeline match in an arbitrary position) *This definition is equal to the previous except that the match does not have to start from the beginning of the source lifeline. Formally,*

$$LLMatch_{\phi}^*(O_l, O_s) \stackrel{\text{def}}{=} \exists O_{beg}, O_{end} \in Occur^* : O_s = O_{beg} \frown \phi(O_l) \frown O_{end}$$

Definition 8 (Sequence diagram match) *Given a LHS sequence diagram L and a source sequence diagram S . The mapping $\phi : L \rightarrow S$ is a sequence diagram match if and only if for all lifelines $l \in L_{LL}$ the following two conditions are satisfied: (1) l is not mapped to a lifeline which is part of an operand, and (2) l is mapped to a lifeline match. Formally,*

$$\begin{aligned} sdMatch_{\phi}(L, S) &\stackrel{\text{def}}{=} \\ \forall l \in L_{LL} : &\phi(l).operand = \text{null} \\ &\wedge \text{ if } l.\text{hasArbEvt} \text{ then } LLMatch_{\phi}^*(l.occs, \phi(l).occs) \\ &\text{ else } LLMatch_{\phi}^1(l.occs, \phi(l).occs) \end{aligned}$$

As pointed out in the previous section, we are allowed to delete combined fragments even though all its spanning lifelines are not explicitly matched. Let Del denote the set of to-be-deleted combined fragments, i.e. $Del = \{\phi(f) \mid f \in (L_F \setminus l(I_F))\}$. The function $delCF(O, Del)$ returns the occurrence sequence O where all combined fragments in Del has been removed.

Definition 9 (Sequence diagram transformation step) *Given a rule $p : L \stackrel{l}{\leftarrow} I \xrightarrow{r} R$, a source sequence diagram S , and a mapping $\phi : L \rightarrow S$, where $sdMatch_{\phi}(L, S)$. The rule p and the mapping ϕ define a transformation step from S to a target sequence diagram T , denoted $S \xrightarrow{p, \phi} T$. The lifelines of T , T_{LL} , are the union of (1) the transformed L lifelines (the occurrences given in an L lifeline are replaced by the occurrences in the corresponding R lifeline (retrieved by the helper function $getOccsR$), (2) all the new R lifelines, and (3) all the unmapped lifelines in S . For each lifeline in the lifeline sets (1) and (3), we need to delete every occurrence that represents a to-be-deleted combined fragment by using the function $delCF$. Formally,*

let $getOccsR(l_i) \stackrel{\text{def}}{=} \text{if } \exists l_i \in I_{LL} : l(l_i) = l_i \text{ then } r(l_i).occs \text{ else } \langle \rangle$

in $S \xrightarrow{p_\phi} T \stackrel{\text{def}}{=} sdMatch_\phi(L, S) \wedge$

$$T_{LL} = \{l_t \mid l_i \in L_{LL} \wedge \exists O_{beg}, O_{end} \in Occur^* : \quad (1)$$

$$\phi(l_i).occs = O_{beg} \cap \phi(l_i.occs) \cap O_{end} \wedge$$

$$l_t.occs = delCF(O_{beg} \cap getOccsR(l_i) \cap O_{end}, Del)\}$$

$$\cup R_{LL} \setminus r(I_{LL}) \quad (2)$$

$$\cup \{l_s \mid l_s \in (S_{LL} \setminus \phi(L_{LL})) \wedge l_s.occs = delCF(l_s.occs, Del)\} \quad (3)$$

7 Related Work

Our methodology is quite similar to the one prescribed by Whittle and Schumann [WS00] and Ziadi et al. [ZHJ04]. Whittle and Schumann need OCL expressions to express similar behavior across multiple diagrams, while we and Ziadi et al. take advantage of the combined fragments which were introduced in UML 2 after the work of Whittle and Schumann.

Ziadi et al. [ZHJ04] define their transformation by pseudocode operating on algebraic definitions of sequence diagrams and state machines, while our transformation is based on graph transformation. Our support for guards in `alt/loop` and support for `par/opt/neg` is new compared to their approach.

Harel et al. [HKP05] define a transformation from Live Sequence Charts to UML state charts, which are described by traditional algorithms. Their focus is on the transformation itself in contrast to our work that provide an improved way to specify such transformations. While we simply ignore negative traces and produce a state machine that does not recognize such behavior, their approach will analyze and detect inconsistency such as defining the same trace as both positive and negative.

Sun [Sun07] specifies a transformation from state charts to state machines in the AToM tool which like our approach takes advantage of combined fragments (`alt` and `loop`). With our fragment operator and the collection operator, we can define the transformation rules completely by graphical models. Sun, on the other hand, needs to use relatively complicated textual pre- and post-conditions associated with the rules.

The MATA tool [WJE⁺09] and Klein et al. [KFJ07] are two promising sequence diagram aspect proposals where transformation on sequence diagrams can be specified based on the concrete syntax and where an occurrence sequence on a lifeline easily can be replaced another occurrence sequence.

The MATA tool also has a way to match combined fragments in a sequence diagram aspect language. However, it is too limited as a basis for the transformation from sequence diagrams to state machines, since there is no way to match a combined fragment with an unknown number of operands.

Klein et al. have no support for matching combined fragments. Furthermore, in Klein et al. all matches are identified and treated at once which is not appropriate for our transformation from sequence diagrams to state machines.

Hermann [Her05] uses algebraic graph transformation, restricted to abstract syntax, to specify transformation rules for sequence diagrams. Without the collection operator and the fragment operator, our transformation rules to state machines will be very difficult to express.

We have not seen other proposals where it is easy to specify that an event or a combined fragment has to be the very first occurrence on a lifeline. Although a

bit cumbersome, it is expressible in other graph transformation approaches by using several NACs.

Our previously defined semantics-based aspect language [GSMPK08] cannot be used as a basis for the transformation from sequence diagrams to state machines, since it is not structure preserving. The structure of combined fragments is utterly important in order to generate readable state machines.

This paper contains some extensions from the conference paper [GMP10]: the modeling process is now described in relation to a refinement theory, a more detailed description of the transformation rules including some flattening rules to produce more optimized state machines, and some details about the mapping from concrete to abstract syntax-based rules.

8 Conclusions

We have shown how concrete syntax-based graph transformation rules can be used to specify a transformation from sequence diagrams to state machines. These rules are much more concise than traditional graph transformation rules which are specified in abstract syntax.

It is a great advantage that the user can specify rules in the well known concrete syntax of sequence diagrams instead of the complicated and less intuitive abstract syntax version. On the other hand, in our approach we need to implement a translation from concrete syntax to abstract syntax-based rules. The extent to which there is a need for sequence diagram transformations in general decides if the implementation effort pays off in practice.

We introduced a novel fragment operator that allows us to graphically specify the matching and transformation of a combined fragment with an arbitrary number of operands. Furthermore, we formalized a suitable way to handle the order of occurrences on a lifeline, which is crucial when specifying transformations of sequence diagrams.

References

- [Bræ08] Bjørn Brændshøi. Consistency Checking UML Interactions and State Machines. Master's thesis, Department of Informatics, University of Oslo, 2008. Available from: <http://urn.nb.no/URN:NBN:no-21036>.
- [GKMP09] Roy Grønmo, Stein Krogdahl, and Birger Møller-Pedersen. A Collection Operator for Graph Transformation. In *Int. Conf. on Model Transformation (ICMT)*. Springer, 2009. doi:10.1007/978-3-642-02408-5_6.
- [GMP08] Roy Grønmo and Birger Møller-Pedersen. Aspect Diagrams for UML Activity Models. In *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Revised Selected and Invited Papers*, volume 5088 of *Lecture Notes in Computer Science*. Springer, 2008. doi:10.1007/978-3-540-89020-1_23.
- [GMP10] Roy Grønmo and Birger Møller-Pedersen. From sequence diagrams to state machines by graph transformation. In *Theory and Practice of Model Transformations, Third International Conference, ICMT*, volume 6142 of *Lecture Notes in Computer Science*. Springer, 2010. doi:10.1007/978-3-642-13688-7_7.

- [Grø09] Roy Grønmo. *Using Concrete Syntax in Graph-based Model Transformations*. PhD thesis, Dept. of Informatics, University of Oslo, 2009. Available from: <http://urn.nb.no/URN:NBN:no-24448>.
- [GSMPK08] Roy Grønmo, Fredrik Sørensen, Birger Møller-Pedersen, and Stein Krogdahl. A Semantics-based Aspect Language for Interactions with the Arbitrary Events Symbol. In *European Conference on Model Driven Architecture – Foundations and Applications (ECMDA)*. Springer, 2008. doi:10.1007/978-3-540-69100-6_18.
- [Her05] Frank Hermann. Typed Attributed Graph Grammar for Syntax Directed Editing of UML Sequence Diagrams. Diploma thesis. Master's thesis, Technical University of Berlin, Department for Computer Science, 2005.
- [HKP05] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *Lecture Notes in Computer Science*. Springer, 2005. doi:10.1007/b106390.
- [HKT02] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In *Graph Transformation, First Int. Conf., ICGT*, 2002. doi:10.1007/3-540-45832-8_14.
- [HMP00] Øystein Haugen and Birger Møller-Pedersen. JavaFrame: Framework for Java-enabled modelling. In *Ericsson Conference on software Engineering (ECSE)*, 2000. Available from: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.107.4635>.
- [KFJ07] Jacques Klein, Franck Fleurey, and Jean-Marc Jézéquel. Weaving multiple aspects in sequence diagrams. *Trans. on Aspect Oriented Software Development*, 3, 2007. doi:10.1007/978-3-540-75162-5_7.
- [KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In *International Workshop on Distributed and Parallel Embedded Systems*, 1999. Available from: <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.113.4291>.
- [LEO06] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Conflict Detection for Graph Transformation with Negative Application Conditions. In *Graph Transformations, Third Int. Conf., ICGT*, Lecture Notes in Computer Science. Springer, 2006. doi:10.1007/11841883_6.
- [Lun07] Mass Soldal Lund. *Operational analysis of sequence diagram specifications*. PhD thesis, Dept. of Informatics, University of Oslo, 2007. Available from: <http://urn.nb.no/URN:NBN:no-18776>.
- [RHS05] Ragnhild Kobro Runde, Øystein Haugen, and Ketil Stølen. Refining UML interactions with underspecification and nondeterminism. *Nordic Journal of Computing*, 2(12), 2005. Available from: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.98.6181>.
- [RHS06] Ragnhild Kobro Runde, Øystein Haugen, and Ketil Stølen. The Pragmatics of STAIRS. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005*, volume 4111 of

- Lecture Notes in Computer Science*, pages 88–114. Springer, 2006. doi:10.1007/11804192_5.
- [Sun07] Ximeng Sun. A Model-Driven Approach to Scenario-Based Requirements Engineering. Master's thesis, School of Comp. Science, McGill Univ., Montreal, Canada, 2007. Available from: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.116.3035>.
- [Tae96] Gabriele Taentzer. *Parallel and Distributed Graph Transformation. Formal Description and Application to Communication-Based Systems*. PhD thesis, Technische Universität Berlin, 1996.
- [Tae03] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Applications of Graph Transformations with Industrial Relevance, Second International Workshop (AGTIVE)*, 2003. doi:10.1007/b98116.
- [WJE⁺09] Jon Whittle, Praveen Jayaraman, Ahmed Elkhodary, Ana Moreira, and João Araújo. MATA: A Unified Approach for Composing UML Aspect Models based on Graph Transformation. *Transactions on Aspect-Oriented Software Development VI. Special Issue on Aspects and Model-Driven Engineering*, 5560, 2009. doi:10.1007/978-3-642-03764-1_6.
- [WS00] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *The 22nd international conference on Software engineering (ICSE)*, 2000. doi:10.1145/337180.337217.
- [ZHJ04] Tewfik Ziadi, Loïc Hérouët, and Jean-Marc Jézéquel. Revisiting statechart synthesis with an algebraic approach. In *26th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2004. Available from: <http://csdl.computer.org/comp/proceedings/icse/2004/2163/00/21630242abs.htm>.

About the authors

Roy Grønmo is a research scientist at SINTEF. He holds a doctor degree in Computer science at the University of Oslo. His main research topics are model-driven development, model and graph transformation, service-oriented modeling and aspect-oriented modeling. Contact him at roy.gronmo@sintef.no, or visit <http://folk.uio.no/roygr>.

Birger Møller-Pedersen is professor at University of Oslo. He has worked with object orientation, from various implementations of SIMULA to the design of BETA. He was a key person in adding object-orientation to ITU SDL (standardized 1992). With Ericsson he contributed to UML2.0 within OMG. Contact him at birger@ifi.uio.no.

Acknowledgments The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project), and by the DiVA project grant no. 215412 (EU FP7 STREP).