Algorithmic Support for Model Transformation in Object-Oriented Software Development

Siegfried Schönberger Rudolf K. Keller Ismail Khriss

Département d'informatique et de recherche opérationnelle
Université de Montréal
C.P. 6128, succursale Centre-ville, Montréal, Québec H3C 3J7, Canada
voice: (514) 343-6782, fax: (514) 343-5834
e-mail: {schoenbe, keller, khriss}@iro.umontreal.ca
http://www.iro.umontreal.ca/~{schoenbe, keller, khriss}

Abstract

Current methods for object-oriented software development provide notations for the specification of models, yet do not sufficiently relate the different model types to each other, nor do they provide support for transformations from one model type to another. This makes transformations a manual activity, which increases the risk of inconsistencies among models and may lead to a loss of information.

We have developed and implemented an algorithm supporting one of the transitions from analysis to design, the transformation of scenario models into behavior models. This algorithm supports the Unified Modelling Language (UML), mapping the UML's collaboration diagrams into state transition diagrams. We believe that CASE tools implementing such algorithms will be highly beneficial in object-oriented software development.

In this paper, we provide an overview of our algorithm and discuss all its major steps. The algorithm is detailed in semi-formal English and illustrated with a number of examples. Furthermore, the algorithm is assessed from different perspectives, such as scope and role in the overall development process, issues in the design of the algorithm, complexity, implementation and experimentation, and related work.

Keywords: Object-oriented analysis and design, model transformation, transformation algorithm, Unified Modeling Language (UML), collaboration diagram, state transition diagram.

1 Introduction

Looking at today's methods for object-oriented software development, we observe that the models produced in the various development activities are only loosely coupled. Most methods describe how to specify models, yet do not sufficiently guide the developer in the task of transforming one model

This work was in part supported by the SPOOL project organized by CSER (Consortium Software Engineering Research) which is funded by Bell Canada, NSERC (National Sciences and Research Council of Canada), and NRC (National Research Council of Canada). Siegfried Schönberger is funded by an Erwin-Schrödinger scholarship, granted by the Austrian FWF - Fonds zur Förderung der Wissenschaftlichen Forschung.

type into another. We see a high need for the detailed description and automated support of such transformations.

Recently, scenario-based development [JCJO92, JK98] has gained a lot of attention. Scenario models provide a rich vocabulary for task-centered behavior description. Scenario models are usually specified during analysis, yet cannot be mapped systematically onto structural and behavioral class descriptions required in design. As part of our work, we have developed an algorithm for transforming scenario models into state transition models.

In defining the subject of our research, we were guided by the fact that certain types of models are more extensively used in one particular development phase than in another. In general, use case models (along with scenario models) are chiefly used during analysis, whereas state transition models are mostly employed during design.

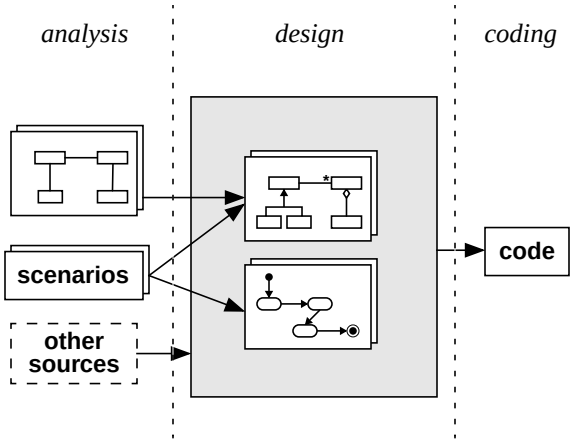

Figure 1: Main information flows in the object-oriented development process.

Figure 1 shows the main sequence of models in which information travels during the analysis, design and coding phases of an object-oriented software development process. Even though such a process will often loop back in an iterative way, with information flowing from later stages back to earlier ones, the principal flow direction is *forward*. Scenario diagrams (instances of use cases), henceforth also called “scenarios”, and other early specification documents are particularly useful in analysis. Class diagrams (specifying the structure of classes and objects; depicted in the figure as a graph of rectangular nodes) are used in analysis and subsequently in design, where they are further refined. State transition diagrams (specifying the behavior of objects; depicted in the figure as a graph of oval nodes) are mostly employed in design.

Class diagrams and state transition diagrams resulting from the design activity constitute the main documents from which a system will be implemented. Scenarios as typical analysis instruments are no longer considered for implementation. Otherwise, the job of translating design documents into code would grow into a highly complicated and tedious task considering the extensive overlaps and redundancies among scenarios themselves, and between scenarios and class and state transition diagrams. However, this procedure bears the risk that invaluable information contained in the scenarios is not taken into account for implementation. Therefore, we have to assure that all relevant information from scenarios is reflected in class and state transition diagrams.

The *Unified Modeling Language (UML)* [RMH+97] constitutes an attempt to unify and standardize modern object-oriented modeling notations. Given the UML’s growing importance as a standard for object-oriented modeling [Col97], we decided to base our work upon it. However, we do not mean to imply that the concepts put forth in this paper exclusively apply to the UML, but rather to object-oriented modeling languages in general, such as *Booch* [Boo94], *Fusion* [CAB+94], *OML* [RHSL96], *OMT* [RBP+91], *OOSE* [JCJO92], *OSA* [EKW92],

Shlaer-Mellor [SM88, SM92], and *VOM* [Sch94].

We have implemented our algorithm and successfully applied it to a number of small and medium-sized models. We suggest that CASE tool builders support the systematic transition from analysis to design models, by implementing algorithms such as ours into their respective tools.

In Section 2 of this paper, we give a brief overview of the UML. Section 3 describes in more detail the diagram types that are relevant for our work. Section 4 gives an overview of the transformation algorithm and discusses all its major steps. In Section 5, we situate the algorithm in the overall development process, assess it from different perspectives, and relate it to the work of others. Section 6, finally, points out ongoing and future work and provides some concluding remarks. Appendices A and B present and explain the grammars of collaboration diagrams and state diagrams, respectively. These grammars are fundamental for the transformation algorithm presented in this paper.

2 Object-Oriented Modeling with the Unified Modelling Language

Object-oriented modeling methods consist of two major parts: a notation as the underlying language for describing the problem and its solution domain, and a process describing the steps from an intended system to an implemented system. Most contemporary methods focus on the phases of analysis and design. They put less emphasis on requirements specification and implementation. Moreover, they barely support the transitions from requirements specification to analysis and from design to implementation.

In every young and immature field it takes some time until the community comes to agree upon common concepts and standards. As for object-oriented modeling notations, with the introduction of the UML a consensus seems to become visible.

The UML [RMH+97] is based on the unification of the syntactic notations, the semantic models and the diagrams of Booch's modeling method [Boo94, Boo96], Rumbaugh's OMT [RBP+91] and extensions thereof [Rum95c, Rum95a, Rum95b], as well as Jacobson's use case modeling [JCJO92]. Further influences on the UML include Fusion's operation descriptions and message numbering [CAB+94], Embley's singleton classes and composite objects [EKW92], and Meyer's pre- and post-conditions [Mey97]. The UML covers only the first aspect of a modeling method, the notation, but does not provide a modeling process. There are, however, ongoing attempts to address process issues in the realm of the UML [Rat98]. The focus of the UML are analysis and design tasks.

UML models are represented as diagrams, which are views of the elements of the underlying models. Diagrams are the primary means for the manipulation of models. A model element may appear in various diagrams.

The UML defines eight types of diagrams, of which the following five types are relevant for our work:

- class diagrams
- use case diagrams
- sequence diagrams
- collaboration diagrams
- statechart diagrams

A *class diagram* represents the logical structure of a system. The elements of a class diagram are classes and objects, together with relationships between these elements. A class features attributes, operations and properties which further characterize the class. Relationships between classes capture asso-

ciations, aggregations and inheritance. Further concepts supported by class diagrams include parameterized classes, utilities, and composite classes.

Use case diagrams represent generic descriptions of an entire transaction involving several objects, together with external actors which interact with the system.

Sequence diagrams and collaboration diagrams are collectively called *interaction diagrams*. An interaction diagram describes an interaction pattern among objects in a system, focusing on a specific function. This is achieved by providing constructs for message exchange and for control flow such as iteration and conditionality. By restricting general interaction patterns, specific instances (scenarios) of use cases can be captured, showing individual executions of a system. These scenarios help the developer in understanding the general collaboration patterns among objects.

Sequence diagrams show interactions among a set of objects, emphasizing the temporal flow of behavior. They are similar to extended Z.120 message sequence charts [ITU94]. *Collaboration diagrams* also show interactions among a set of objects. In addition, they detail the relationships among the objects at hand. However, they do not show time as a separate dimension, nor do they accommodate all the timing expressions found in sequence diagrams. In collaboration diagrams, sequence numbers are used to show the sequence of messages and concurrent threads.

Statechart diagrams, the UML term for *state transition diagrams* henceforth used in this paper, show the evolution of an object over time by means of states and transitions (cf. [Har87, HG96]). A state represents a period of time in an object's life cycle. Upon entry and exit of a state, an instantaneous operation can be executed. While in a state, activities that take some time to complete can be executed. A state can also have its proper state variables. In order to convey high-level and detailed-level views, a state can be decomposed into substates, with the possibility of expressing concurrency.

Generally, a transition is a response to an external event received by an object which is in a certain state. A transition is instantaneous; that is, it takes zero time to execute from the model's point of view. The specification of a transition includes the event that triggers the transition, guard conditions that must be fulfilled so the transition can be executed, events to be sent to other objects, instantaneous operations on the object, and timing marks. Further concepts supported by statechart diagrams are event hierarchies, branching and merging of concurrent threads, stubbed transitions and history states.

3 Framework for the Transformation Algorithm

In the work presented, we focus on the UML's collaboration diagrams and statechart diagrams, which, among other types of diagrams, lie at the core of analysis and design, respectively. For brevity, we will refer to them in the rest of this paper as *CollDs* and *StateDs*, respectively. CollDs and sequence diagrams as defined by the UML are roughly equivalent in terms of expressiveness. For our work, we chose collaboration diagrams because the UML documentation describes features such as conditionality, iteration, concurrency mechanisms, and link types in much more detail than the corresponding features of sequence diagrams. As target diagrams, we chose state transition diagrams rather than activity diagrams because the former focus on flows driven by external events, as opposed to activity diagrams, which focus on internal events. As CollDs

primarily show the interaction *between* objects, the choice of StateDs is evident.

For a precise description of our transformation algorithm, a formal definition of both CollDs and StateDs is required. Therefore, we use EBNF-grammars for these two types of diagrams (see Appendices A and B, respectively). The UML-style formalization of CollDs and StateDs in [RMH+97, Figures 9 and 10] notwithstanding, we decided to introduce an EBNF-grammar, in order to gain conciseness and expressiveness.* Our grammar does not cover the visual aspects of the diagrams, i.e., boxes, lines and the like are not included as symbols, because they are not relevant for the mechanics of the transformation algorithm under discussion.

3.1 Collaboration Diagrams

The grammar presented in Figure 12 in Appendix A is derived from the UML documentation set [RMH+97]. We have decided to limit ourselves in this work to the fundamental concepts of CollDs, thus introducing a few simplifications. Note that these simplifications do not compromise the expressive power of CollDs in any substantial way. They are detailed in Appendix A.

For sample CollDs, refer to Figures 7, 9, and 11, which have been adapted and extended from [RMH+97, Figures 37 and 40]. The example illustrated in Figures 7 and 9 will serve as a running example for the rest of this paper.

Figure 7 shows operation `displayPositions()`. The purpose of this operation is to draw a line, consisting of several segments, in a given window. Each segment is defined by two points (beads). The object that performs this operation is `Wire`. Figure 9 is an extension of Figure 7, in which predecessor messages (cf. Appendix A) have been added.

Figure 11 depicts operation `start(job)` carried out by the object `FactoryJobMgr`. The purpose of this operation is to have a robot remove some item from an oven whose temperature and doors must be controlled appropriately. This CollD illustrates the use of synchronous and asynchronous messages (depicted by $\rightarrow$ and $\rightharpoonup$, respectively) and concurrent threads (shown by lowercase letters) synchronized by single and multiple predecessors (cf. Appendix A).

3.2 Statechart Diagrams

The formal syntax of StateDs presented in Figure 14 in Appendix B is derived from the UML documentation set [RMH+97]. It covers only those aspects we depend on in our work. Concepts that are irrelevant to our discussion such as timing or history states have been omitted from the grammar, others, such as constructs for describing synchronous events, have been added.

The notion of *events*, as defined by the UML for specifying object interactions in StateDs, is restricted to asynchronous data transmission, which is inherently unidirectional. Bidirectional information flows are not supported. CollDs, on the other hand, support synchronous events, that is, messages can have return values (cf. Figure 12 in Appendix A). Hence, to accommodate these features, a transformation algorithm such as ours requires transitions in StateDs to be able to send synchronous events and receive results. The necessary extensions are explained in Appendix B and reflected in the grammar for StateDs shown in Figure 14 in Appendix B.

Sample StateDs are presented in Figures 8 and 10. Figure 8 shows the result of applying our algorithm to the CollD in Figure 7,

* For instance, `a|b` cannot be adequately expressed in the UML-style formalization; textual annotations have to be used instead.

whereas Figure 10 depicts one of the StateDs generated from Figure 9.

4 Transformation Algorithm

In this section, we introduce our algorithm for transforming a CollD into StateDs. Refer to Figures 7 and 8 for an illustration of the transformation process.

We go on the simplifying assumption that no StateDs exist when the transformation process is started. Rather, they are built up from scratch during transformation. Note that CollDs and StateDs are quite often developed in parallel. A general transformation algorithm would have to accomplish the more complex task of integrating information originating from several CollDs into already existing StateDs. This issue is further discussed in Section 5.2 *Transformation Issues.*

In the discussion below, we will refer to objects in CollDs as *home objects* and as *linked objects*. Given a link, its home object is the object from which the link is originating, and the linked object is the object towards which the link is pointing.

Below, we first provide an overview of the algorithm. Next, we describe the algorithm in detail, using semi-formal English. Then, the algorithm's fifth and final step, the *Sequencing of Transitions*, which is the most interesting one, is detailed and illustrated. The section ends with the discussion of a postprocessing step, the *Compressing of Statechart Diagrams*.

4.1 Overview of the Algorithm

The transformation of a CollD into StateDs is a multi-step process consisting of:

(1) Creating empty StateDs
(2) Creating state variables
(3) Creating transitions for home objects
(4) Creating transitions for linked objects
(5) Sequencing transitions

Step (1) creates a StateD for every distinct class implied by the objects in the CollD. Step (2) introduces as state variables all variables that are not attributes of any of the objects of the CollD. This concerns all the variables used as return values of messages, as well as those referred to in iteration and conditional expressions.

Applied to the example in Figure 7, Step (1) creates five empty StateDs (`Controller`, `Wire`, `Line`, `Bead`, and `Window`). Step (2) introduces `i`, `n`, `line`, `r0`, and `r1` as state variables for `Wire`, and `window` for `Line` (see Figure 8).

Step (3) creates transitions for the objects from which messages are sent (home objects). This consists of setting the `event`-parts, creating auxiliary transitions for transitions that are waiting for more than one external event, setting the `sendClause`-parts, and copying temporary data for use in Step (5). Step (3) is organized into the twelve substeps (3.1) to (3.12). Note that the created transitions are not complete from the viewpoint of a StateD. At this stage in the transformation process, they are not yet embedded into a network of states and transitions; that is, they still lack their `fromNode` and `toNode` information.

Applied to the example in Figure 7, Step (3.1) creates the transitions identified by the following `sendClause`-events: `displayPositions` in `Controller`; `position`, `create`, and `display` in `Wire`; and `add` in `Line`. `DrawSegment`, being a message to `"self"`, is not transformed into the StateD of `Wire` (Step (3.2)). Steps (3.3) and (3.4) are skipped, as there are no messages with one or more predecessors in the example. Steps (3.5) and (3.6) determine the `syncIndicator` and the `returnValue` of the identified transitions: → `displayPosition` in `Controller`; → `r0:= position`, → `r1:=`

position, → create, and → display in Wire; and → add in Line. Step (3.7) sets the target of the sendClause of the identified transitions: wire for displayPositions in Controller; Line for create, beads(i-1) and beads(i) for position, and line for display in Wire; and window for add in Line. Step (3.8), concerning messages with multiple predecessors, is not applicable in this example. Step (3.9) sets the parameters for the sendClause events, resulting in: displayPositions(window) in Controller; position(), create(r0, r1), and display(window) in Wire; and add(self) in Line. Steps (3.10), (3.11), and (3.12) copy sequence numbers, recurrence information as well as "new" flags into the appropriate transitions for use in Step (5): recurrence information in Wire which will lead to the creation of action /initLoop, and the "new" flag into transition Line.create in Wire (see Figure 8).

Step (4) creates transitions for the objects to which messages are sent (linked objects). This consists of creating the event-part and returnValue-part of a transition.

Applied to the example in Figure 7, Step (4) creates transitions redisplay() in Controller; displayPositions(window) in Wire; create(r0,r1) and display(window) in Line; position() in Bead; and add(aLine) in Window (see Figure 8).

The goal of Step (5), the final step of the algorithm, is to bring for all StateDs the set of generated transitions into correct sequences. This involves the creation of the (empty) states, split bars and merge bars necessary for the proper connection of all the transitions. As a result of this step, the generated transitions are connected to states, that is, their fromNode and toNode attributes are well-defined.

Applied to the example in Figure 7, Step (5) creates an initial state and a regular state in Line and connects them by transition create(r0, r1). Furthermore, transition display(window) is connected with both its fromNode and toNode being the regular state. StateDs Controller, Wire, Bead, and Window are sequenced as well. Note that Wire comprises a cycle of transitions as well as a pair of concurrent transitions, as a result of the iteration instruction and the concurrent messages, respectively, in the example in Figure 7.

4.2 The Algorithm in Detail

The transformation algorithm takes a CollD as input. Its output is a set of StateDs. In the semi-formal notation used in this section, we rely on the following conventions:

symbol>subsymbol:
refers to subsymbol of symbol, assuming that subsymbol can be directly reached according to the formal definitions of CollDs (Figure 12) and StateDs (Figure 14), respectively. If any of symbol or subsymbol is not specified in a given context, symbol>subsymbol is considered unspecified.
Example: object>className

symbol>{subsymbol}:
shortcut notation for the set of all subsymbols in symbol.
Example: event>{argument}

"+" sign: string concatenator.

Step (1): Creating empty StateDs
For each object in CollD>{object} do

(1.1): if (StateD for object does not exist), then
create new StateD for object;
StateD>name:= object>className.

Step (2): Creating state variables
For each link in CollD>{object}>{link} create state variables for links of type "local" (2.1), for undeclared result objects (2.2), and for undeclared variables in iterations and conditions (2.3):

(2.1): If `link>linktype = "local"`, then
create new state variable `stVar` in `StateD` of `object`;
`stVar>variableName:= link>role>roleName;`
`stVar>className:= link>linkedObject>className.`

(2.2): For each `message` in `link>{message}` do
if `message>returnValue` not declared in (class diagram or StateD) of object, then
create new state variable `stVar` in `StateD` of `object`;
`stVar>variableName:= returnValue.`

(2.3): For each `var` occurring in `(link>{message}>sequenceExpression>recurrence>clause` do
if `var` not declared in (class diagram or `StateD)` of `object`, then
create new state variable `stVar` for `object`;
`stVar>variableName:= var>variableName.`

Step (3): Creating transitions for home objects

For each message in `CollD>{object}>{link}>{message}` do (3.1) to (3.11):

(3.1): Create a new transition `trans` in `StateD` of `object`.

(3.2): If `object ≠ link>linkedObject` (sender and receiver objects must be different), then do (3.3) to (3.9). (3.3) and (3.4) deal with the `event`-part, and (3.5) to (3.9) deal with the `sendClause`-part of `trans`.

event-part of trans (3.3 to 3.4):

(3.3): If the message has exactly one predecessor, then look for message with same sequence number as indicated as predecessor. Its name becomes the event trigger of the transition:
If number of elements in `message>predecessor = 1 then`
find message `msg` in `CollD` where `msg>sequenceNumber = message>predecessor>sequenceNumber;`
if `msg ∉ object>{link}>{message},`then
`trans>event>eventName:= msg>messageName;`
`trans>event>{parameter}:= msg>{argument}.`

(3.4): If the message has more than one predecessor, then look for messages with same sequence numbers as indicated in predecessor. For each of these messages (in case they are not sent by *`object`*) an auxiliary transition `auxTrans` is created. The auxiliary transitions will merge into a synchronization bar (`mergeBar`) which in turn will be followed by the transition corresponding to `message`:
If number of elements in `message>predecessor > 1` then
for each `sN` in `message>predecessor>{sequenceNumber}`do
find message `msg` in `CollD` with `msg>sequenceExpression>sequenceNumber = sN;`
if `msg ∉ object>{link}>{message},` then
create a new transition `auxTrans` for `object;`
assign a unique name to `eventName:`
`auxTrans>event>eventName:= msg>messageName + msg>sequenceNumber;`
`auxTrans>event>{parameter}:= msg>{argument};`
`auxTrans>tempSeqNumber:= message>sequenceNumber + "-".`

Comment: `tempSeqNumber` as well as `tempRecurrence` and `tempIsNew` (see Steps (3.11) and (3.12) below, respectively) are auxiliary variables introduced to preserve the information necessary for Step (5) of the algorithm. At the end of Step (5), this data is discarded. `tempSeqNumber` is assigned the concatenation of `message>sequence-Number` and "-", indicating that the `auxTrans` transitions must be ordered such that they immediately precede the transition whose sequence number is `sequenceNumber`, that is `trans`, the transition corresponding to

message.

sendClause-part of trans (3.5 to 3.9):

(3.5): Map asynchronous messages onto asynchronous events, and procedure call and flat message types onto synchronous events (the UML documentation is not specific in this respect):
trans>sendClause>syncIndicator:=
  message>controlFlowType.

(3.6): If message>returnValue is specified, then
  trans>sendClause>result:=
    message>returnValue.

(3.7): Define trans>sendClause>target. Send the first event to the class of the linked object (constructor), and any other event to the object designated by rolename:
If (link>role>"new" is specified or link>role>"transient" is specified) and message has the lowest sequence number of all messages sent from object to linkedObject, then
  trans>sendClause>target:=
    link>linkedObject>className;
  elsif link>role>roleName is specified, then
    trans>sendClause>target:=
    link>role>roleName;
  else target remains empty (to be specified during design).

(3.8): Define eventName. In case *sendClause>event* is one out of several events for which *link>linkedObject* is waiting simultaneously (specified by a message with multiple predecessors), the sequence number of *message* is appended to *eventName*. In this way, the events for which *link>linkedObject* is waiting carry unique names that match the unique names produced in Step (3.4). Otherwise, eventName is defined as message> messageName:
If there is a message msg in link>linkedObject>{link}>{message} for which number of elements in msg> predecessor > 1 and (message> sequenceExpression>sequenceNumber ∈ msg>predecessor), then
  trans>sendClause>event>eventName:=
    message>messageName +
    message>sequenceExpression>
    sequenceNumber;
else
  trans>sendClause>event>eventName:=
    message>messageName.

(3.9): trans>sendClause>event> {parameter}:= message>{argument}.

Temporary data of trans (3.10 to 3.12):

(3.10): trans>tempSeqNumber:= message>sequenceNumber.

(3.11): trans>tempRecurrence:=message> sequenceExpression>recurrence.

(3.12): For each link in CollD>{object}> {link} do
  if link>role>"new" is specified, then
    trans>tempIsNew:= "new".

Step (4): Creating transitions for linked objects

(4.1): Create a new transition representing the start message. Only name, arguments, result, and sequence number of the start message are considered; all other information is ignored:
Create a new transition trans in StateD of startMessage>object.
trans>event>eventName:=
  startMessage>messageName;
trans>event>{parameter}:=
  startMessage>{argument};
trans>tempSeqNumber:=
  startMessage>sequenceNumber;
if startMessage>returnValue is specified, then trans>returnValue:=
  startMessage>returnValue.

Comment: Giving back the returnValue in this transition is a design decision made by our algorithm (cf. Section 5.2 *Transformation Issues, Hard-coded Design Decisions*).

(4.2): Create transitions for messages being

sent to *object>link>linkedObject* that are sequential, i.e., messages with no predecessor, and that are not sent from *object,* i.e., no self-messages. Note that transitions for linked objects for concurrent messages have already been created in Step (3.4):

For each message in CollD>{object}>{link}>{message} do
  if message>sequenceNumber ∉ link>linkedObject>{link}>{message}>predecessor} and object ≠ link>linkedObject, then
    create a new transition trans in StateD of object>link>linkedObject;
    trans>event>eventName:= message>messageName;
    trans>event>{parameter}:= message>{argument};
    trans>tempSeqNumber:= message>sequenceNumber;
    if message>returnValue is specified, then
      trans>returnValue:= message>returnValue.

Step (5): Sequencing Transitions

This step brings all transitions generated in Steps (3) and (4) into correct sequences, connecting them by states, split bars and merge bars.

(5.1): Create a transition list from StateD>{transition} that is ordered by tempSeqNumber. Transitions whose tempSeqNumber comprises a trailing '-' are ordered such that they immediately precede the one transition comprising the same tempSeqNumber except for the trailing '-'. These transitions reflect multiple predecessors of a message (cf. Step (3.4)).

(5.2): Create a regular state curState.

(5.3): If TempIsNew of the first transition in transition list is specified, create an initial state. Connect the initial state with the curState via the first transition in transition list. Note that TempIsNew indicates that the transition is a constructor of the object.

For each transition in transition list do (5.4) and (5.5):

(5.4): Determine type of transition: sequential, iteration, or thread.

(5.5): If type of transition is sequential, then

  lastToNode:= curState.toNode;
  create a regular state curState;
  connect lastToNode.toNode to curState.fromNode via transition;

elsif type of transition is iteration,

  then sequence transitions as described in Section 4.3 *Illustration of Step (5), Iteration;*

  elsif type of transition is thread,

    then sequence transitions as described in Section 4.3 *Illustration of Step (5), Conditionality and Concurrency.*

4.3 Illustration of Step (5)

In this section, we address the transformation of four particular message types: iteration messages, conditional messages, concurrent messages and messages with multiple predecessors. Discussing these aspects of the algorithm in isolation and illustrating them with examples may help in understanding overall functioning.

Iteration

A message with an iteration indicator ("*", *iteration message*) is transformed by placing all messages that belong to the iteration message into a loop of states and transitions.

For example, the list of messages below is transformed into the StateD in Figure 2.

Messages
1.1 msg1()
1.2 * [loopVar:=lowerBnd..upperBnd]: msg2()
1.2.1 msg3()
1.2.2 msg4()
1.3 msg5()

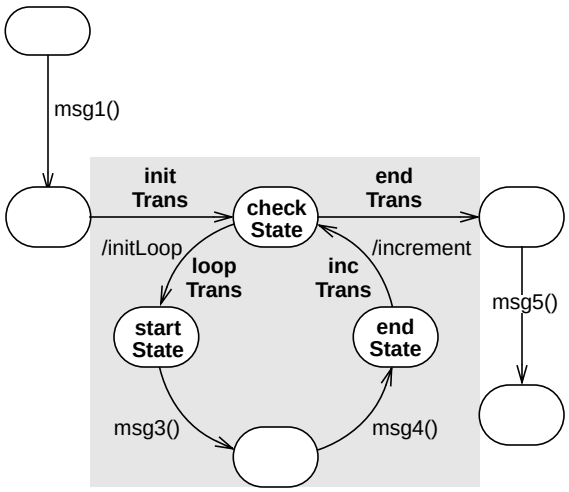

Figure 2: Transformation of sample iteration message.

Let (1.2) be the transition derived from the message with sequence number 1.2. Then, the iteration specification of message msg2() is temporarily stored in transition (1.2) (in `tempRecurrence`; see Step (3.11) of the algorithm). In Step (5), the shaded area of Figure 2 is generated which substitutes transition (1.2) and msg2(), and which is embedded between the two transitions stemming from msg1() and msg5(). Message msg2() is transformed into transitions initTrans, loopTrans, incTrans, and endTrans, and into states checkState, startState, and endState. The messages belonging to the iteration message (msg3() and msg4()) are transformed into the loop's body.

Step (5) uses the information in `tempRecurrence` of transition (1.2) to check if it indicates an iteration ("`*`") and to set the following conditions:

[loopVar <= upperBnd] in loopTrans
[loopVar > upperBnd] in endTrans

An iteration message in context, together with its transformation, can be found in Figures 7 (→ 1.1*[i:=1..n]: drawSegment(i)) and 8 (StateD of class Wire), respectively.

An iteration message that is sent to another object is transformed essentially in the same way as a message that is sent to `"self"` (see example in Figure 7). The entire loop infrastructure (initTrans, loopTrans, incTrans, and endTrans) is generated for the object that sends out the iteration message. All messages that are subordinate to the iteration message are transformed in the same way as regular messages.

Conditionality and Concurrency

We make the observation that conditional and concurrent messages are quite similar in nature. Both represent a branch of control emerging from a state. The difference is that concurrent branches are executed simultaneously whereas conditional branches are mutually exclusive, i.e., only one of them is executed. Therefore, the algorithm handles conditional and concurrent messages in similar ways.

We define a *conditional message group* as a group consisting of a message with a conditional recurrence clause *(conditional message)* and of all its subordinate messages. We transform such a group into sequences of transitions and place them into concurrent substates. One concurrent substate is created for every conditional message group. The first transition of each of these sequences emerges from a common split bar (short heavy line). Similarly, the last transition of each of these sequences joins a common merge bar (short heavy line). Each conditional message group is placed into a concurrent substate.

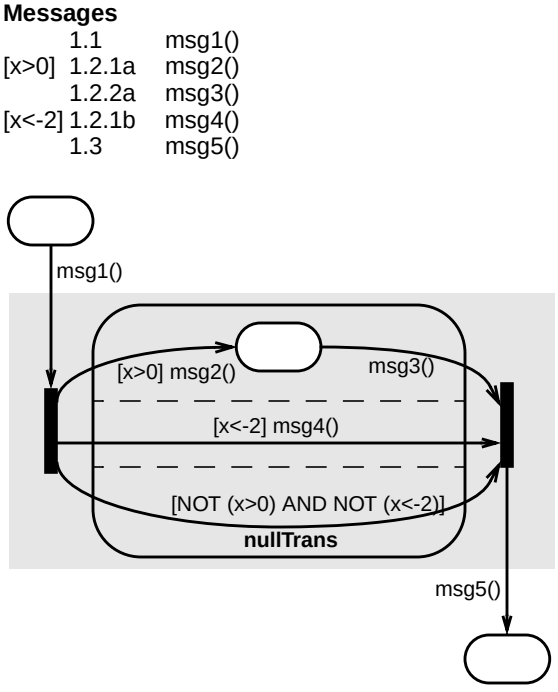

Figure 3: Transformation of sample conditional messages.

Figure 3 outlines how a group of sample conditional messages and their subordinate messages is transformed into a StateD. The shaded area, embedded between transitions msg1() and msg5(), shows the result of the transformation of "threads" a and b.

The transition `nullTrans` is executed if none of the conditions of the conditional messages holds. The `condition`-part of `nullTrans` is constructed by negating all the conditions of the conditional messages and logically combining them with `AND`.

A group of conditional messages in context, together with their transformation, can be found in Figures 9 (→ 1.1a [window.display=#on] : drawSegment, → 1.1b [window.display=#off] : flashBorder()) and 10, respectively. Note that, in contrast to the drawSegment message shown in Figure 7, the drawSegment message of Figure 9 does not represent an iteration.

We transform messages that belong to a concurrent thread (*concurrent messages*) into sequences of transitions and place them into concurrent substates. One concurrent substate is created for each different thread indicator. The first transition of each of these sequences emerges from a common split bar (short heavy line). Similarly, the last transition of each of these sequences joins a common merge bar (short heavy line). Each thread is placed into a concurrent substate.

Figure 4 shows the transformation of a sample set of messages forming two concurrent threads into a StateD. There are two initial messages 1.2.1a and 1.2.1b, distinguished by their thread indicators a and b, which spawn the two concurrent threads 1.2.1a - 1.2.1a.1 - 1.2.1a.2 and 1.2.1b - 1.2.1b.1, respectively. The shaded area, embedded between transitions msg1() and msg7(), shows the result of the transformation of these threads.

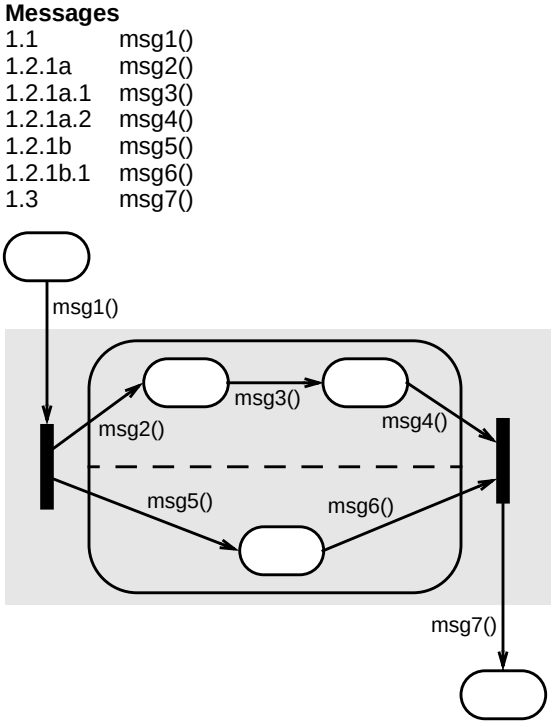

Figure 4: Transformation of sample concurrent threads.

For examples within context, refer to the concurrent messages of Figures 7 (→ 1.1.1a: r0:=position(), → 1.1.1b: r1:=position()) and 9

(→ 1.1a.1a: r0:=position(), → 1.1a.1b: r1:=position()), and their transformations in Figures 8 (StateD of class Wire) and 10, respectively.

Handling conditionality in the same way as concurrency, i.e., as concurrent substates, might appear as an overly expensive approach. However, in order to obtain efficient code from a StateD with conditionality, a code generator would simply place the execution of the first branch whose condition evaluates to true onto the main thread. In case there are further branches whose conditions evaluate to true, those branches would be placed onto additional threads.

Multiple Predecessors

Predecessors of a message are specified in `message>predecessor`. They indicate sequence numbers of other messages that must have been sent before the message can be sent. For each element in a message's `predecessor` field (designating a message sent by a foreign object*), a new transition to a subsequent synchronization bar (merge bar) is created. The only specified part of such a transition is `event`, which is set to the message name of the message whose sequence number corresponds to the respective sequence number in `predecessor`. To distinguish homonymous messages, the event name is concatenated with the sequence number of the respective message.

* Currently, the algorithm does not handle certain types of predecessors of messages, such as predecessors that refer to messages on a parallel thread of the same object.

Messages

```
                1.7a msg1()
                1.6b msg1()
[1.7a,1.6b]  2     msg2()
```

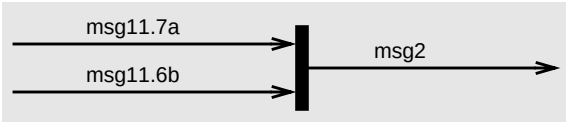

Figure 5: Transformation of multiple predecessors of sample message.

Figure 5 exemplifies this transformation scheme. [1.7a,1.6b] 2 msg2() is the message to be transformed into a StateD. As there are two predecessors 1.7a and 1.6b, two transitions msg11.7a and msg11.6b are created which subsequently merge into transition msg2. The transitions referring to messages 1.7a and 1.6b are required to send events msg11.7a and msg11.6b, respectively.

An example of a message with multiple predecessors can be found in Figure 11 (→ [1.1.2b.2a, 1.1.2b.3b] 1.1.3a: takeMaterial). Its transformation into a StateD has not been reproduced in this paper.

4.4 Compressing Statechart Diagrams

As a postprocessing step, a transformation algorithm may implement various techniques for obtaining StateDs with a reduced number of states and transitions. In its current form, our algorithm implements the following two techniques:

- *merge an event-only-transition with a subsequent sendClause-only transition*
  If a transition contains only sendClauses (and/or actions and/or a returnValue) and its preceding transition contains only an event (and/or a guard condition), with the intermittent state being empty, then these two transitions are collapsed into one, thus eliminating the state that has connected them. For an example, refer to the StateD of class Wire in Figure 8. The tran-

sition displayPositions(window) /initLoop is the result of applying this technique to the two original transitions displayPositions(window) and /initLoop.

- *eliminate duplicate transitions*
  If two transitions that connect the same states contain identical events, guard conditions, actions, sendClauses, and returnValues ("duplicate transitions"), then one of them is removed. For an example, refer to the StateD of class Bead in Figure 8. Applying this technique to the two original transitions position() ^r0 and position() ^r1 results in the single transition position() ^r0. Note that the transitions in this example lead back to the same state from which they originate.

We have identified a number of further techniques, which are currently not implemented in our algorithm, including:

- *move sendClauses into a state*
  If a transition connecting two empty states contains only sendClauses (and/or actions), then the states that are connected by this transition are collapsed into one, and the sendClauses (and/or actions) become actions of this state. For an example, refer to the StateD of class Wire in Figure 8. Applying this technique to transition → line := Line.create(r0,r1) would collapse its connected states into one, and the transition would become the action of that state.
- *collapse sequences of sendClause-only transitions into single transitions*
  A sequence of sendClause-only (and/or action-only) transitions, with the intermittent states being empty, is collapsed into one single transition comprising the list of all sendClauses (and/or actions) involved in the sequence. For an example, refer to the StateD of class Wire in Figure 8. Applying this technique would move the two sendClauses → line := Line.create(r0,r1) and → line.display(window) along with the action increment into one single transition comprising all three of these items.

5 The Algorithm in Perspective

The transformation algorithm presented in this paper performs the transformation from CollDs to StateDs, one of the key transitions from analysis to design. In this section, we first place this particular transformation into the landscape of the overall development process, addressing a number of other transformations that need to be performed. Next, we assess our algorithm from a number of perspectives, addressing issues such as completeness and extensibility. Then, we discuss its complexity as well as our implementation and experiences. Finally, we review related work.

5.1 Transformations in the Development Process

The information flows in the object-oriented development process as indicated in Figure 1 suggest a number of transformations. Recall that in the UML, scenarios are captured in interaction diagrams, which can be either sequence diagrams or CollDs. The following transformations need to be performed:

- sequence diagrams to class diagrams,
- sequence diagrams to StateDs,
- CollDs to class diagrams,
- CollDs to StateDs, and
- class diagrams and StateDs to code.

The transformation from interaction diagrams to a class diagram consists of two parts. First, all associations between objects in the interaction diagrams must be properly reflected in the class diagram, either as attributes of the respective classes or as class variables. Second, all the messages that an object receives must be added to the operations compartment of the object.

The transformation from interaction diagrams to StateDs is more complex. Scenarios

typically show several objects interacting with each other. The challenge is to pick all the right pieces of information that concern one specific class and build a StateD for this class.

After design has been completed, class diagrams and StateDs are transformed into code. This transformation is comparatively straightforward. The distribution of structural and behavioral information in class diagrams and StateDs largely corresponds to the distribution in code: classes and objects in class diagrams and StateDs are transformed into classes and objects in code. A class diagram describes a number of classes. A StateD describes one class. These diagrams can directly be mapped into appropriate units of code.

Note that the complete development process involves more transformations than those depicted in Figure 1. For instance, in early analysis, use case models might be produced with the ensuing need of transforming them into the analysis models of Figure 1. Another example is the transformation of model parts within one particular development phase, e.g., from class diagrams to StateDs. Furthermore, in the context of an iterative development process, the transformation from design models back to analysis models becomes of particular interest. Finally, when taking a system architecture point of view, further views and transformations may come into play, as described in [Kru95].

5.2 Transformation Issues

The issues addressed in this section concern the transformation from CollDs to StateDs. However, due to their general nature, they should be considered in most other types of transformations as well.

Incompleteness

The transition from interaction diagrams to class diagrams and StateDs is not a one-to-one mapping process. StateDs describe a system in a general way, thus holding more information than interaction diagrams, which are employed to show specific scenarios. It is important to see that the transformation process only yields a first yet fundamental draft of a StateD for subsequent refinement. During work on StateDs, detailed design information that cannot be derived from scenarios is added.

Due to the fact that StateDs potentially contain more information than CollDs, some parts of StateDs cannot be filled during the transformation process and remain empty. An example are states: our algorithm produces nameless states because CollDs provide no information for assigning meaningful names. Another example are actions: Actions describe the internal behavior of an object. As CollDs show the external behavior of objects (interactions), they do not provide any data that can systematically be used to specify actions of transitions or states. Note that an exception to this latter example are the “pseudo-actions” initLoop and increment which result from the transformation of iteration messages (cf. Figure 2).

Architectural considerations are another element that is generally not reflected in analysis models. A system’s architecture (with respect to issues such as communication mechanisms or inheritance relationships) is built during design, adding fundamental information to design models. Such information can usually not be derived from sequence diagrams and CollDs.

Uninteresting StateDs

During design, StateDs of certain classes may be discarded. Often, such classes merely provide services that do not require a

specific sequence of their functions. In this case, state transition diagrams are not necessary. Keep in mind, however, that an uninteresting StateD in one application domain may well be of interest in another domain.

Hard-coded Design Decisions

Once the complete information in a CollD is carried over into StateDs, there is some freedom in constructing the individual StateDs. This is mainly due to the different viewpoint that is assumed in a StateD (component-centered in a StateD versus task-centered in a CollD). In designing the algorithm, we were guided by our desire for simplicity and readability. At various points, the resulting StateDs reflect design decisions that are encoded in the algorithm.

For instance, a message in a CollD may wait for a result. There is more than one possibility of transforming this setting into a StateD.

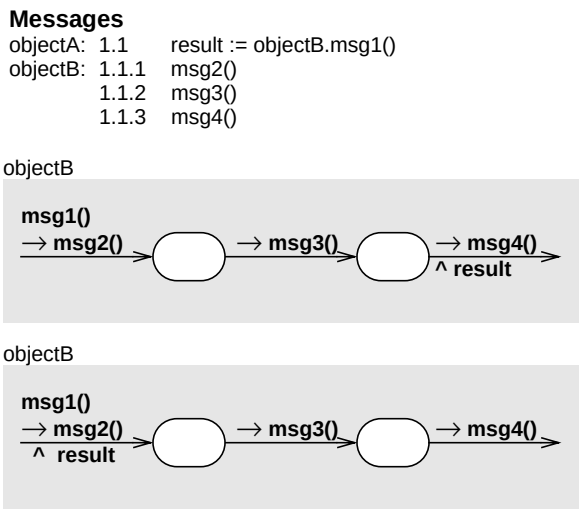

Figure 6: Two possibilities of transforming a message.

Consider the four messages in Figure 6, belonging to two different objects, objectA and objectB. msg1 is sent by objectA, whereas msg2, msg3, and msg4 are sent by objectB. Figure 6 also shows two possible StateDs for objectB.

The StateD shown in the upper part reasons that all messages that are subordinate (in terms of their sequence number) to a message that is waiting for a result have to be sent before the result can be returned to the sender. In our example, objectA sends msg1 to objectB, waiting for result. objectB waits for event msg1, then sends msg2, msg3, and msg4, before returning result to objectA.

The lower StateD shows an alternative transformation of these messages. objectB waits for event msg1, then sends msg2 and returns result. Only thereafter does it send msg3 and msg4.

Based on the information contained in CollDs, it is difficult to decide at which point a result should be returned. Our algorithm implements the latter approach.

Another example of a hard-coded design decision is the way we deal with multiple predecessors (see Section 4.3 *Illustration of Step (5), Multiple Predecessors*). Finally, the techniques discussed in Section 4.4 *Compressing Statechart Diagrams* also constitute hard-coded design decisions.

Extensibility

It should be noted that our algorithm can easily be extended to cover the grammatical concepts of CollDs we left out for simplicity (cf. Appendix A), and to cope with future extensions of UML CollDs. This claim is substantiated by the fact that we have developed the algorithm in an incremental way, initially supporting only subsets of CollDs and StateDs. Moreover, we had to update the algorithm on several occasions, in order to comply with the current version 1.1 of the UML, departing from version 0.8. These various updates could be done with only limited effort, although some of the changes affecting CollDs and StateDs had been quite substantial.

Transforming multiple CollDs and integrating StateDs

In the presented algorithm, StateDs that may already exist for a given class are not taken into account. However, a more general algorithm should also accomplish the task of merging newly generated information with already existing StateDs. This is required for two reasons. Firstly, there are typically a number of CollDs describing operations of one class. The resulting StateDs overlap and therefore have to be merged. Secondly, in practice, CollDs and StateDs are often developed in parallel. Again, the algorithm has to merge existing with new information. Extending the algorithm into this direction is the subject of ongoing work (see below).

5.3 Complexity of the Algorithm

For the discussion of the worst-case complexity C of the algorithm, let N_o be the number of objects in the CollD, N_l the number of links, and N_m the number of messages. Since the algorithm consists of 5 steps, C is the sum of C_1, C_2, C_3, C_4, and C_5, providing that C_i represents the complexity of Step (i). Step (1) creates a StateD for every distinct class in the CollD. C_1 is therefore $O(N_o)$. Step (2) creates state variables based on information from links, as well as from the return values, iteration expressions, and conditional expressions found in messages. C_2 is therefore $O(\max(N_o, N_l, N_m))$. Step (3) creates transitions for all the objects from which messages are sent, so there are no more than N_m messages to be processed. The processing of each message is $O(N_m)$, since this step includes a method that tries to find for each message a message with a given sequence number (see Step (3.4) in Section 4.2 *The Algorithm in Detail*). C_3 is therefore $O(N_m^2)$. The same applies to Step (4) which creates transitions for all the objects to which messages are sent. C_4 is therefore $O(N_m^2)$. Step (5) puts all transitions into sequence. A transition list is used whose size does not exceed N_m. Sequencing first sorts the transition list and then processes each transition of the list. The sorting algorithm has a complexity of $O(N_m * \log_2 N_m)$. The processing of a single transition involves finding other transitions in the transition list and is $O(N_m)$. Therefore, the processing of all transitions is $O(N_m^2)$, and C_5 is $O(N_m^2)$. Consequently, the overall worst-case complexity of the algorithm is $O(N_m^2)$.

5.4 Implementation of the Algorithm and Experiences

Our algorithm has been implemented as a system of 15 Java classes and some 2100 lines of code. We defined a textual input format for CollDs and a textual output format for StateDs.

Not surprisingly, the implementation revealed several errors in the original version of the algorithm. It also helped us in adapting and validating the algorithm for the various UML versions, in that tedious manual walkthroughs could be replaced by sessions with the interactive Java debugger. Moreover, it facilitated the development of the compression techniques considered for the postprocessing step of the algorithm.

The algorithm was tested on a number of small and medium-sized examples, with the most complex CollDs comprising 10 objects and some 50 messages. For all test cases, the runtime performance of the algorithm was excellent (less than a second). As test cases, all the examples described in the UML documentation set [RMH+97] were used, as well as examples known from the literature, for instance, the traffic-light controller presented by Rumbaugh et al. [RBP+91] and further discussed in [KM94], and an extended ver-

sion of the library application described in [EP98].

All example CollDs presented in this paper were processed by our algorithm. The diagrams of the resulting StateDs were drawn manually, based on the textual descriptions generated by the algorithm.

5.5 Related Work

The need for algorithmic support for transformations between different development stages has long been recognized. However, there has been little work for addressing this issue in object-oriented development. Below, we will discuss relevant work in this and related domains.

Arguably one of the first groups addressing program synthesis from example traces are Biermann and Krishnaswamy [BK76]. They have developed an algorithm that takes as input sample sequences of primitive actions (like assignments) and assertions and generates a corresponding program.

Koskimies and Mäkinen, leveraging off the work by Biermann and Krishnaswamy, devise a synthesis algorithm for state transition diagrams [KM94], henceforth referred to as KM-algorithm. The KM-algorithm is tailored to OMT [RBP+91] which they extend by "algorithmic scenario diagrams." These diagrams follow the notation of OMT's event trace diagrams, but also comprise graphical notations for conditionality and repetition, as well as constructs for assertions and subscenarios. Note that these extended scenario diagrams can be expanded into simple scenario diagrams. Furthermore, they introduce actions and states into their scenario diagrams in order to facilitate the reverse engineering part of their work, that is, the transition from OMT state diagrams back to scenario diagrams.

The expressive power of UML CollDs, the input to our algorithm, is similar to that of their algorithmic scenario diagrams, with the notable exception that CollDs also support concurrency, and less importantly, multiple predecessors and synchronicity aspects. CollDs comprise neither actions nor states, and we refrained from introducing these or any further concepts in order to respect the focus of scenario diagrams: scenario diagrams are meant to merely show the exchange of messages between objects, whereas state transition diagrams are meant to capture both the internal and external behavior of an object [RBP+91, RMH+97].

The KM-algorithm synthesizes scenario diagrams into OMT state diagrams. UML StateDs, the output of our algorithm, are very similar to OMT state diagrams. In contrast to the KM-algorithm, our algorithm makes extensive use of the concurrency constructs of UML StateDs, in order to process concurrency as found in UML CollDs.

Being rooted in the algorithm of Biermann and Krishnaswamy [BK76], the KM-algorithm uses backtracking to keep the number of generated states at a minimum. Our algorithm does not guarantee a minimum number of states; however, we have developed a number of techniques for StateD compression (see Section 4.4 *Compressing Statechart Diagrams*). Due to backtracking, the KM-algorithm has exponential complexity in the worst case. In contrast, the complexity of our algorithm is polynomial. The KM-algorithm is incremental, allowing for the merging of scenario diagrams into previously generated state transition diagrams, whereas our algorithm is not incremental yet (see below). Finally, the two algorithms differ in some of the design decisions taken. As an example, the KM-algorithm maps some events of scenario diagrams to state actions and others to transitions; our algorithm transforms them a priori into sendClauses associated with tran-

sitions, and only post-processing may convert them into state actions.

Koskimies and his group have developed the *SCED* tool [KMST96, KSTM98], which currently supports state transition diagram synthesis ("design by example") based on the KM-algorithm as well as "design by animation." At the core of design by animation is the symbolic execution of state transition diagrams and the generation of execution traces in the form of scenario diagrams. Note that this approach is being successfully applied in other domains, for instance, in real-time system development with SDL [OFMP+94], where tools such as SDT [Tel95] allow for the interactive simulation and validation of SDL behavior specifications and the recording of the simulation and validation runs as message sequence charts [ITU94]. Design by animation in SCED allows the developer to complement scenarios by playing the role of a missing object and/or by specifying events to be sent by existing objects. Specifically, each time when SCED is unable to continue the symbolic execution of an incomplete state transition diagram, it prompts the developer for input. Such input not only leads to more comprehensive scenario diagrams, but allows SCED to complete existing state transition diagrams and to generate new state transition diagrams. As a consequence of its design by animation feature, SCED can also be used for certain reverse engineering tasks (see [KSTM98]).

SCED certainly qualifies as a blueprint for a CASE tool supporting our algorithm. It provides graphical editors for scenario diagrams and state transition diagrams, and implements the two design techniques mentioned above. SCED demonstrates how transformation algorithms, once implemented and embedded in an interactive environment, may lead to powerful design tools.

In the realm of object-oriented development, the algorithmic work of Koskimies et al. comes probably closest to ours. It should be stressed that the value and strength of such an algorithmic transformation approach may heavily depend on the particular context. Whereas Koskimies et al. and ourselves strongly advocate the transformation of interaction models into behavior models, Bordeleau and Buhr, for instance, explicitly refrain from such an approach in their *UCM-ROOM* design method [BB96]. Their method provides guidance for manual mapping of use case maps (UCMs), which capture high-level system behavior, into Z.120 message sequence charts [ITU94], which in turn may be mapped into ROOM system structures [SGW94]. ROOM system structures are sets of actors whose communication is governed by contracts. According to Bordeleau and Buhr, the expressiveness of their intermediate representation, the Z.120 message sequence charts, is insufficient to warrant the generation of "good actor behavior models."

In certain domains, for instance in real-time system development based on synchronous specification and programming languages such as LUSTRE [HLR92], the role of state transition models may be quite different from that in object-oriented development. Rather than being an important instrument for system design, they may become mere implementation artifacts. Using synchronous languages, systems are specified as objects that communicate via events and messages. Such specifications, being more generic and more comprehensive than the scenario models normally produced in object-oriented analysis, allow for direct code generation. Ironically, the generated code typically implements quite complex state transition diagrams. Recall, however, that scenario models in object-oriented development do not capture general system behavior. Therefore, direct code generation from scenario

models does not make sense; rather, general behavior models such as represented in state transition diagrams must be built to move towards implementation.

The idea of transforming trace and scenario diagrams is also being pursued in domains other than object-oriented development. Somé et al. [SDV96], for instance, devise such a synthesis algorithm for requirements engineering. Their algorithm builds upon operation semantics (preconditions, postconditions, and "withdrawn" conditions), and generates parts of system specifications based on information from scenario descriptions and related domain-specific information. Another example from the domain of requirements engineering is the work of Hsia et al. [HSG+94]. They formalize a user view (corresponding to several scenarios) as a scenario tree which they transform into a regular grammar and subsequently into a state transition diagram. Note that in their approach, the states of the resulting diagram are manually identified by the analyst when building the scenario tree. Finally, as an example from the telecommunications domain, the research by Ichikawa et al. [IIK+91] may be mentioned. They introduce a language for message sequence description as a basis for protocol synthesis. Such descriptions may automatically be transformed into descriptions of communicating processes that implement the sequences.

6 Conclusion and Future Work

The algorithm presented in this paper addresses one of the key transformations in object-oriented development, the transformation of scenario models into behavior models. It is embedded in the UML, the emerging standard notation for object-oriented analysis and design. The algorithm supports the transformation of CollDs into StateDs, covering all major concepts of CollDs.

A prototype implementation of the algorithm and the subsequent test on a number of small and medium-sized examples makes us believe that the algorithm is fit for provision in CASE tools. We therefore suggest tool builders to implement it, together with other, complementary transformation algorithms (see Section *5.1 Transformations in the Development Process*), in their tools.

An area of ongoing work is the extension of our approach for accommodating multiple CollDs as input and integrating existing StateDs. Rather than extending the algorithm presented in this paper, we decided to apply it as is to each of the CollDs given as input. Notably, we do not want the developer to clutter the input CollDs with extra information, e.g., state names, just for the sake of subsequent integration. Rather, we have the developer intervene *after* applying our algorithm for tasks such as labelling states and refining the structure of the StateDs. Once the StateDs are processed in this way, an algorithm is applied that generates integrated StateDs, one for each class at hand. This approach, together with a preliminary implementation, is described in [KEK98].

An area of future work is the adaptation of the algorithm for conducting consistency checks. Of interest are both checks that verify the consistency between related scenarios, and checks that verify the consistency between scenarios on the one hand, and StateDs on the other hand. Clearly, the provision of such consistency checks will be highly relevant in tool implementations.

We would like to suggest that a major consideration for designing future modeling methods be the ease of transformation and the coherence among the models of the method. Within the software development community, there is a growing consensus for employing the same modeling notation for analysis as well as for design. However,

given the constant evolution of modeling methods, new concepts are added, such as use cases, distribution and real-time issues, which do not always integrate easily into the notation of the method. To further consistency, traceability, automation, and tool support in software development, method designers should pay special attention to the issues of coherence and ease of transformation.

Acknowledgments

We would like to thank the associate editor and the anonymous referees for their detailed and informed comments and suggestions for improving this paper. We are grateful to Aonix for providing us the *SoftwareThroughPictures* CASE tool, supporting us in the experimental part of our research.

References

[BB96] Francis Bordeleau and Ray J. A. Buhr. *The UCM-ROOM design method: From use case maps to communicating state machines*. Technical report, Carleton University, Ottawa, Canada, September 1996. submitted for publication.

[BK76] Alan W. Biermann and Ramachandran Krishnaswamy. *Constructing programs from example computations*. IEEE Transactions on Software Engineering, 2(3):141–153, March 1976.

[Boo94] Grady Booch. *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1994. Second edition.

[Boo96] Grady Booch. *Object Solutions: Managing the Object-Oriented Project*. Addison-Wesley, 1996.

[BR95] Grady Booch and James Rumbaugh. *Unified method for object-oriented development*, documentation set version 0.8 (white paper). Rational Software Corporation, Santa Clara, CA, 1995.

[CAB+94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, Inc., 1994.

[Col97] Derek Coleman(moderator). *UML: The language of software blueprints?* In Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97), pages 201-205, Atlanta, GA, October 1997.

[EKW92] David W. Embley, B. D. Kurtz, and S. N. Woodfield. *Object-Oriented Systems Analysis - A Model Driven Approach*. Prentice-Hall, Inc., 1992.

[EP98] Hans-Erik Eriksson and Magnus Penker. UML-Toolkit. John Wiley and Sons, 1998.

[Har87] David Harel. Statecharts: *A visual formalism for complex systems.* Science of Computer Programming, 8:231–274, June 1987.

[HG96] David Harel and Eran Gery. *Executable object modeling with statecharts.* In Proceedings of the Eighteenth International Conference on Software Engineering, pages 246–257, Berlin, Germany. IEEE, 1996.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. *Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE*. IEEE Transactions on Software Engineering, 18(9):785-793, September 1992.

[HSG+94] Pei Hsia, Jayarajan Samuel, Jerry Gao, David Kung, Yasufumi Toyoshima, and Cris Chen. Formal approach to scenario analysis. IEEE Software, 11(2):33-41, March 1994.

[IIK+91] Haruhisa Ichikawa, Masaki Itoh, June Kato, Akira Takura, and Masashi Shibasaki. *SDE: Incremental specification and development of communications software.* IEEE Transactions on Computers, 40(4):553–561, April 1991.

[ITU94] ITU. *Z.120 MSC. Message Sequence Charts*, 1994. Geneva, Switzerland.

[JCJO92] I. Jacobson, M. Christerson, P. Jonson, and G. Overgaard. *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison-Wesley, 1992.

[JK98] Matthias Jarke and Reino Kurki-Suonio (editors). *Special issue on scenario management*. IEEE Transactions on Soft-

ware Engineering, 24, 1998. Expected publication date: Fall 98.

[KEK98] Ismail Khriss, Mohammed Elkoutbi, and Rudolf K. Keller. *Automating the synthesis of UML statechart diagrams from multiple collaboration diagrams*. Technical Report GELO-84, Université de Montréal, Montreal, Quebec, Canada, March 1998.

[KM94] Kai Koskimies and Erkki Mäkinen. *Automatic synthesis of state machines from trace diagrams.* Software — Practice & Experience, 24(7):643–658, July 1994.

[KMST96] Kai Koskimies, Tatu Männistö, Tarja Systä, and Jyrki Tuomi. *SCED: A tool for dynamic modelling of object systems.* Technical Report A-1996-4, Department of Computer Science, University of Tampere, 1996.

[Kru95] Philippe B. Kruchten. *The 4 + 1 view model of architecture*. IEEE Software, 12(6):42-50, November 1995.

[KSTM98] Kai Koskimies, Tarja Systä, Jyrki Tuomi, and Tatu Männistö. *Automated support for modeling oo software*. IEEE Software, 15(1):42-50, January/February 1998.

[Mey97] Bertrand Meyer. Object-Oriented Software Construction. Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1997.

[OFMP+94] Anders Olsen, Ove Faergemand, B. Moller-Pedersen, R. Reed, and J. R. W. Smith. *Systems Engineering using SDL-92*. North-Holland, 1994.

[Rat98] Rational Software Corporation. *Rational O bjectory Process 4.1 - your UML process*, February 1998. Santa Clara, CA. Available at <http://www.rational.com/support/techpapers/>.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

[RHSL96] G. Rasmussen, Brian Henderson-Sellers and G. C. Low. *An object-oriented analysis and design notation for distributed systems.* Journal of Object-Oriented Programming, 9(6): pages 14-27, October 1996.

[RMH+97] Rational Software Corporation, Microsoft, Hewlett-Packard, Oracle, Sterling, MCI, Unisys, ICON, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum, Ptech, Taskon, Reich Technologies, Softeam. *UML Notation Guide, version 1.1*. Rational Software Corporation, Santa Clara, CA,

[Rum95a] James Rumbaugh. *OMT: The dynamic model.* Journal of Object-Oriented Programming, 8(2): pages 6–12, February 1995.

[Rum95b] James Rumbaugh. *OMT: The functional model.* Journal of Object-Oriented Programming, 8(3): pages 10–14, March 1995.

[Rum95c] James Rumbaugh. *OMT: The object model.* Journal of Object-Oriented Programming, 8(1): pages 21–27, January 1995.

[Sch94] Siegfried Schönberger. *VOM - Visual Object Modelling*. Ph.D. thesis, Johannes-Kepler-University, Linz, Austria, 1994.

[SDV96] Stéphane Somé, Rachida Dssouli, and Jean Vaucher. *Toward an automation of requirements engineering using scenarios.* Journal of Computing and Information (JCI), 2(1):1110–1132, January 1996.

[SGW94] Bran Selic, Garth Gullekson, and Paul Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[SKS97] Reinhard Schauer, Rudolf K. Keller, and Siegfried Schönberger. Extending state diagrams for higher expressiveness and more general applicability. Technical Report DIRO-1065, Université de Montréal, Montreal, May 1997.

[SM88] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis. Modeling the World in Data*. Yourdon Press Computing Series, 1988.

[SM92] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles: Modeling the World in States*. Prentice-Hall, Inc., 1992.

[Tel95] TeleLOGIC, Malmo, Sweden. *SDT 3.0 Reference Manual*, 1995.

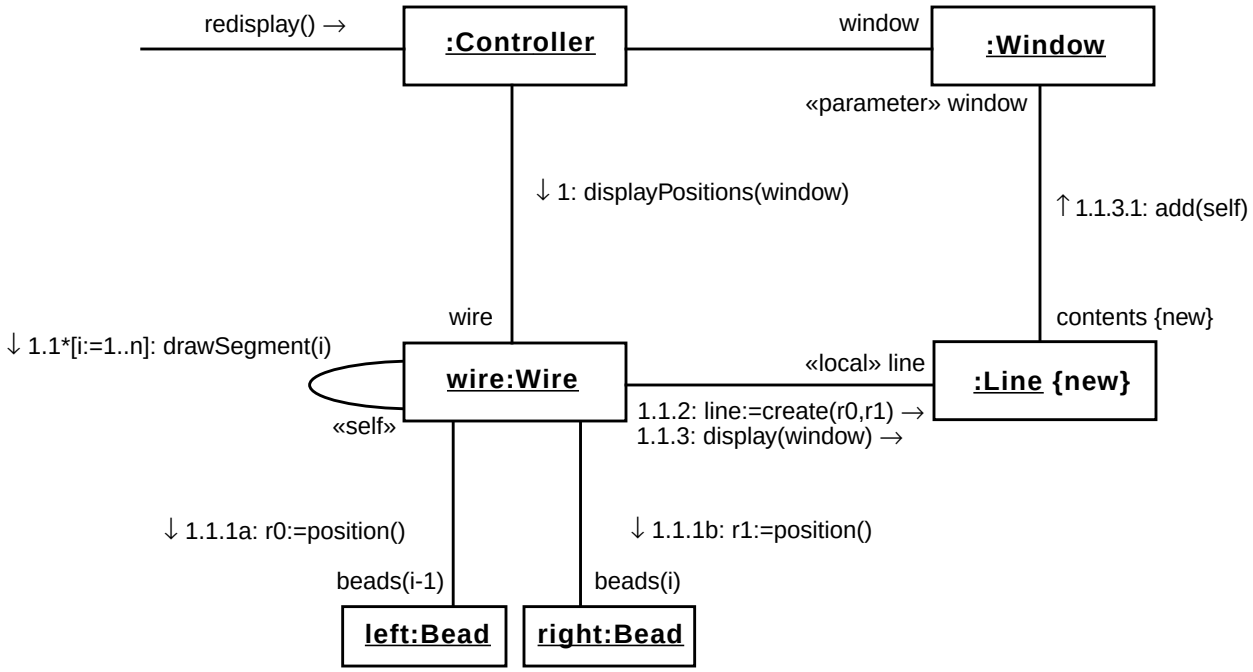

Figure 7: Sample collaboration diagram (adapted from [RMH+97], Figure 37).

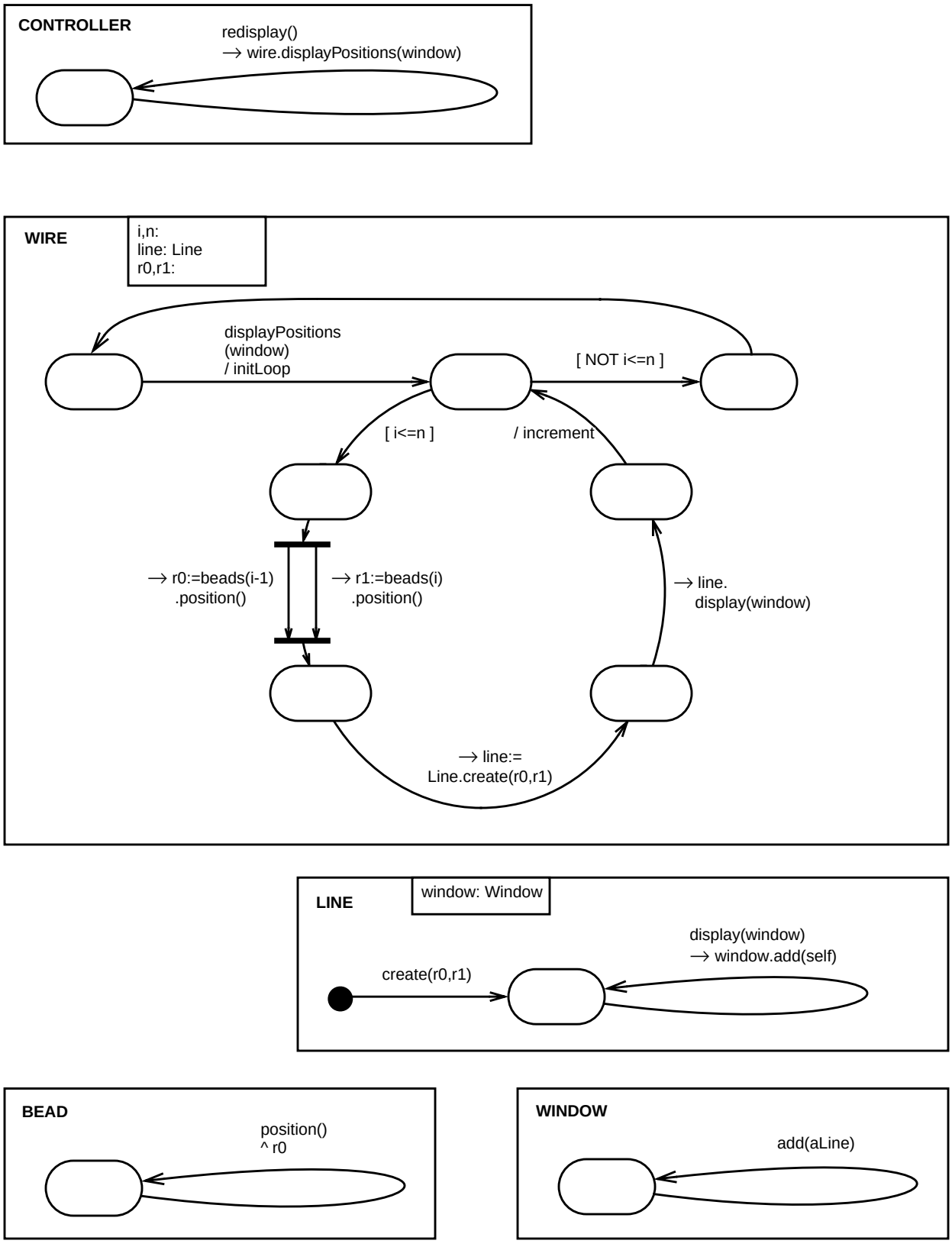

Figure 8: State diagrams resulting from transformation process applied on collaboration diagram in Figure 7.

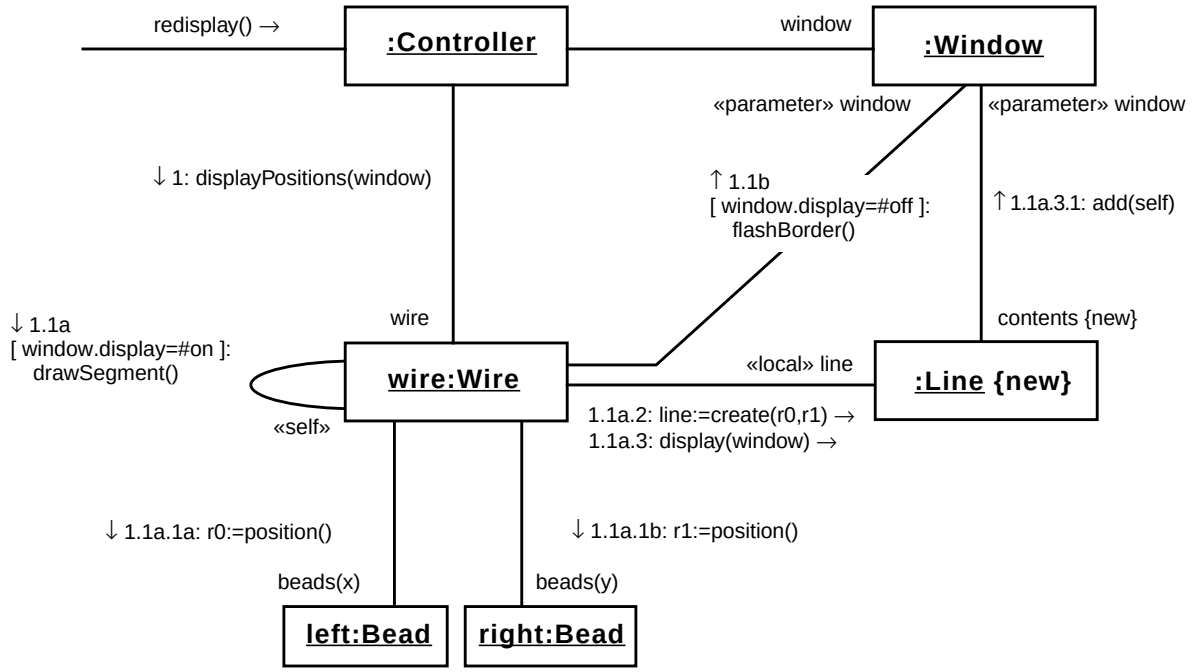

Figure 9: Sample collaboration diagram with conditional messages (extension of example in Figure 7).

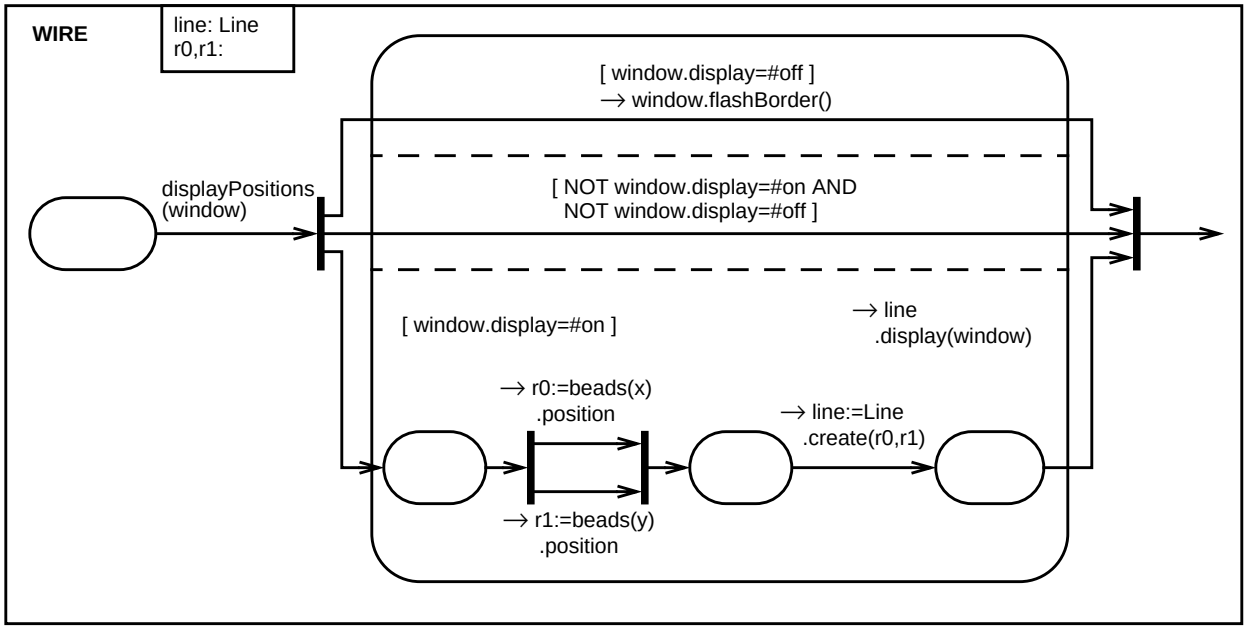

Figure 10: State diagram for class `Wire` resulting from transformation process applied on collaboration diagram in Figure 9.

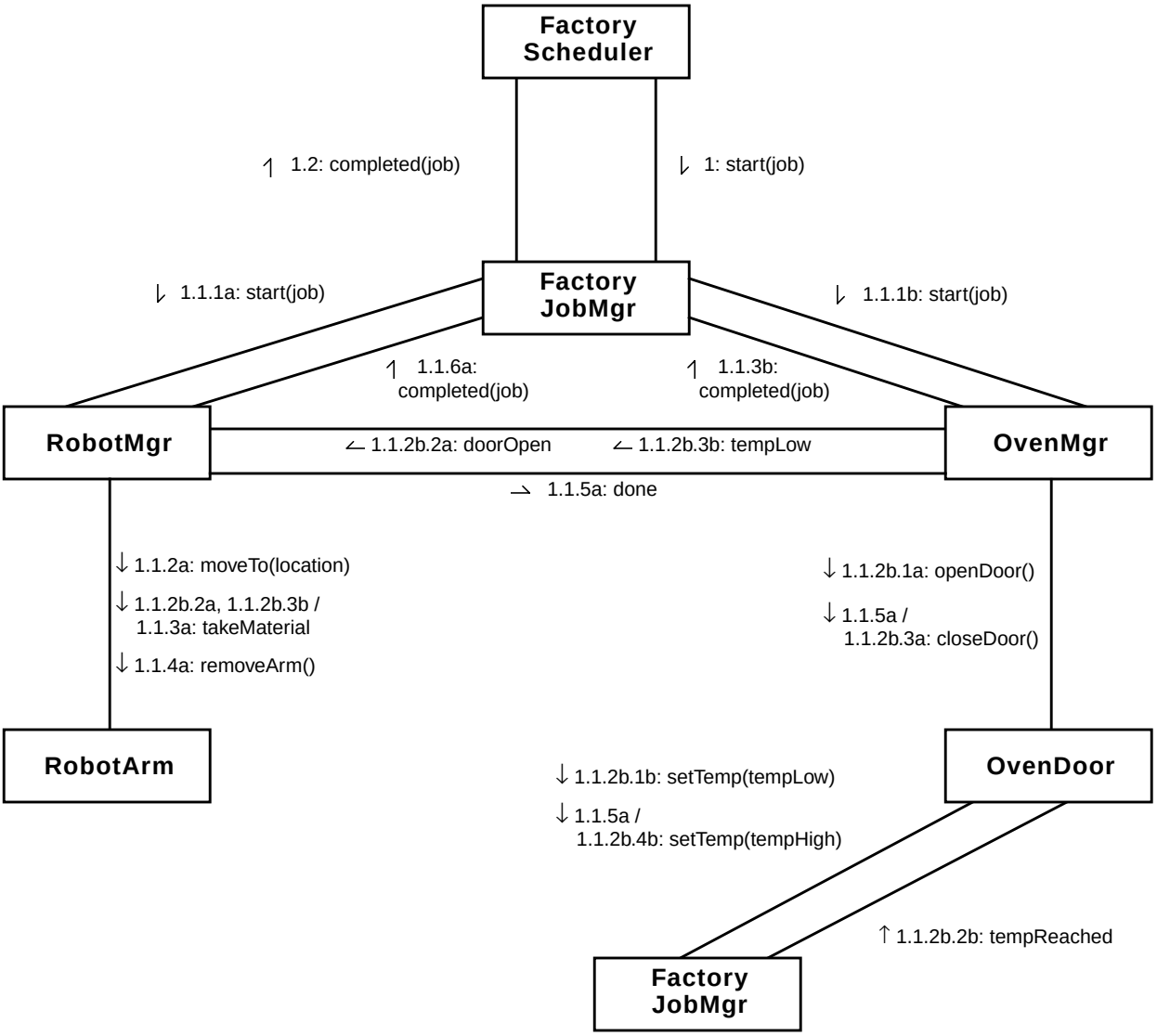

Figure 11: Sample collaboration diagram comprising synchronous and asynchronous messages and concurrent threads synchronized by single and multiple predecessors (adapted and extended from [RMH+97], Figure 40).

App. A: Grammar of Collaboration Diagrams

Figure 12 shows the grammar for collaboration diagrams, as derived from the UML documentation set [RMH+97].

```
CollD = startMessage {object}.
startMessage = message object.
object =
   ["new" | "destroyed" |
   "transient"]
   [objectName] [":" packageName]
   [":" className]
   {attributeName "=" value}
   {link}.
link =
   linkedObject [role] {message}.
linkedObject =
   <reference to object>.
role =
   ["new" | "destroyed" |
   "transient"] roleName.
linkType =
   "association" | "global" |
   "local" | "parameter" | "self".
message =
   controlFlowType
   [predecessor]
   [sequenceExpression]
   [returnValue ":="]
   messageName ["("
   [ argument {"," argument} ] ")"].
controlFlowType =
   procedureCall | flatFlow |
   asynchronousFlow.
predecessor =
   sequenceNumber
   {"," sequenceNumber } "/".
sequenceExpression =
   sequenceNumber [recurrence] ":".
sequenceNumber =
   integer {("." integer) | char }.
recurrence =
   ["*" ["||"] ] clause.
argument =
   <format not prescribed by UML>
clause =
   <format not prescribed by UML>
```

Figure 12: Grammar for collaboration diagrams.

The symbols `objectName`, `packageName`, `className`, `attributeName`, `roleName`, `returnValue`, `messageName`, and `sequenceName` are all strings. They are introduced as separate symbols for better readability.

The vocabulary provided by CollDs for specifying an operation consists of three basic elements: *objects*, *links*, and *messages*. Objects are connected to each other by links. Messages can be attached to links to show communication between objects.

An *object* may show the following details:

- name of the object
- class of the object
- list of attributes

A *link* may show the following details:

- linked object
- role
- list of messages

A link is a unidirectional connection between two objects. Several messages can be attached to a link, all of which are sent to the linked object. If the modeling problem requires a bidirectional connection between two objects, two links - one in each direction - have to be established.

A full-fledged *message* may show the following details (see Figure 13):

- control flow type ($\rightarrow$ procedure call, $\rightarrow$ flat (synchronous), or $\rightharpoonup$ asynchronous message)
- predecessor (to indicate sequencing, as for instance in Figure 11)
- sequence expression
- recurrence (to indicate iteration or conditional message)
- return values
- message name and arguments

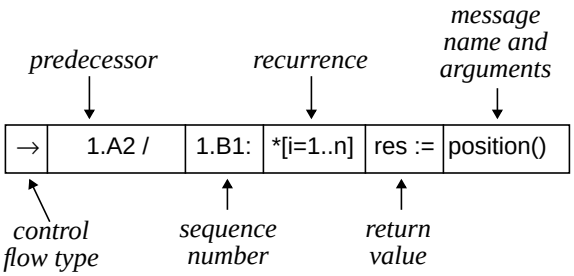

Figure 13: Structure of a sample message in a collaboration diagram.

Limiting ourselves in this work to the fundamental concepts of CollDs, we have introduced a few simplifications. For one thing, we do not support timing marks. Likewise, multiplicity indicators for objects within an enclosing composite class are not handled. Distribution issues like location, migration and dynamic classification of objects are left out as well.

The diagram from which Figure 7 has been adapted [RMH+97, Figure 37], shows qualifiers and rolenames separate from each other. However, for simplicity, we decided to merge qualifiers into rolenames. Figure 7, for instance, shows an association between `Wire` and `Bead` which is qualified with `i-1`. This means that an object `Wire`, together with the qualifier `i-1`, refers to an object `Bead`. Usually, an associated object is being made known to an object by a rolename. In our example, rolename `beads` is assigned to a *set* of objects of class `Bead`. Therefore, to uniquely identify one particular object, we merge rolename and qualifier into `beads(i-1)`. In conclusion, we presume that a rolename contains all pertinent information.

App. B: Grammar of Statechart Diagrams

The grammar of StateDs (Figure 14) is derived from the UML documentation set [RMH+97]. It comprises only those aspects on which we depend in our work.

```
StateD =
   className
   {stateVariableDecl}
   {transition}.
stateVariableDecl =
   stateVariable ":" [className]
   ["=" initValue].
fromNode, toNode =
   <reference to node>.
node =
   state | splitBar | mergeBar.
state =
   initialState |
   regularState |
   terminalState.
regularState =
   [name]
   {stateVariableDecl}
   ["entry" action]
   ["do" action]
   ["exit" action]
   {subState}.
transition =
   fromNode
   [event]
   [guardCondition]
   {"/" action}
   {sendClause}
   ["^" returnValue]
   toNode.
subState = StateD.
event =
   eventName "("
   [parameter {"," parameter}] ")".
guardCondition =
   "[" booleanExpression "]".
sendClause =
   syncIndicator
   [result ":="]
   target "." event.
syncIndicator =
   "→" | "⇀".
target =
   className | objectName.
action =
```

```
actionName "("
[argument {"," argument}] ")".
```

Figure 14: Grammar for statechart diagrams.

The symbols `className`, `stateVariable`, `eventName`, `result`, `objectName`, and `actionName` are all strings. The symbols `initValue`, `returnValue`, and `argument` are all expressions in the sense of programming languages.

Local variables can be used at the top level of a StateD. They are also called *state variables*, as the whole StateD is contained in a nameless state. Thus, a StateD consists of a name, a set of state variables, a set of nodes, and a set of transitions.

The `splitBar` and `mergeBar` symbols indicate splitting and merging of concurrent threads, respectively. They are represented as short heavy lines that are connected to states and transitions (for examples, see Figures 4, 5, 8, and 10).

As CollDs define different kinds of control flow types, we distinguish between synchronous and asynchronous events in StateDs as well. To this end, we propose a synchronicity indicator [Sch94, SKS97]: synchronous events are indicated with a full arrowhead →, whereas asynchronous events are shown with a half arrowhead ⇀. This notation is based on the control flow type notation of CollDs. It replaces the ^ indicator, which merely serves as a separator in the original syntax of StateDs.

In the synchronous case (in the asynchronous case, ⇀ replaces →), transitions may take on the following form:

```
event
  [ guardCondition ]
  / action
  → result := target.event
  ^ returnValue
```

`result` specifies where the result of a synchronous event should be stored. `result` may either be a state variable or an attribute of the object at hand. Note that `result` can only occur in the synchronous case.

Introducing constructs that allow an object to send a synchronous event and to receive a result requires appropriate constructs for the object which is to receive this event. We therefore provide, as an additional element for the specification of a transition, the `returnValue`, together with the preceding delimiter "^". The receiver may set `returnValue` from within `action`, by sending a synchronous event to a third object, or by simply indicating an attribute of the object.

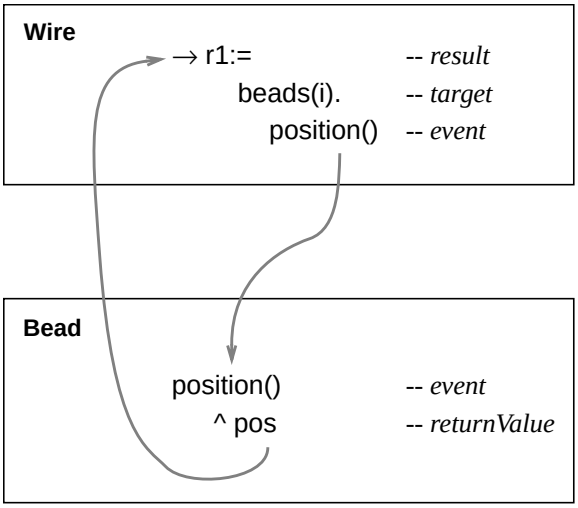

Figure 15: Sending an event in a statechart diagram (excerpt from Figure 8).

Figure 15 illustrates how a synchronous event is sent and a result is received. Sender Wire sends event position() to receiver beads(i) which is of class Bead. Event position() triggers a transition in the receiver, and beads(i) returns its attribute pos in response. This example is drawn from Figure 8 and is further discussed in Section 4 *Transformation Algorithm*.