Declarative techniques for model-driven business process integration

J. Koehler
R. Hauser
S. Sendall
M. Wahler

Business process integration and automation are among the most significant factors driving the information technology industry today. In addressing the manifold technology challenges of integration and automation, new standardization efforts aim at improving the interoperability of businesses by moving toward a declarative specification of business processes, that is, one which describes what a business process does and not how it is implemented. At the same time, Model Driven Architecture® focuses on improving the software-engineering methods with which business process solutions are implemented by separating the business or application logic from the underlying platform technology and representing this logic with precise semantic models. In this paper, we present an approach to the model-driven generation of programs in the Business Process Execution Language for Web Services (BPEL4WS), which transforms a graphically represented control-flow model into executable code by using techniques that originated in compiler theory. We discuss the underlying algorithms as well as general questions concerning the representation and analysis of model transformations. We study a declarative representation of transformation rules, where preconditions and postconditions are represented in the Object Constraint Language. By adopting a declarative approach, we pave the way for future automatic consistency checking of transformation rules and bidirectional reconciliation of evolving models.

Model Driven Architecture** (MDA**) has been proposed by the Object Management Group (OMG) to enhance the efficiency and quality of software development and to reinforce the use of an enterprise architecture strategy. Models can be specified from different views, such as that of a business analyst or an information technology (IT) architect, and can be represented at different levels of abstraction. MDA separates the business or application logic from the underlying platform technology and represents this logic with precise semantic models. In particular, MDA distinguishes between

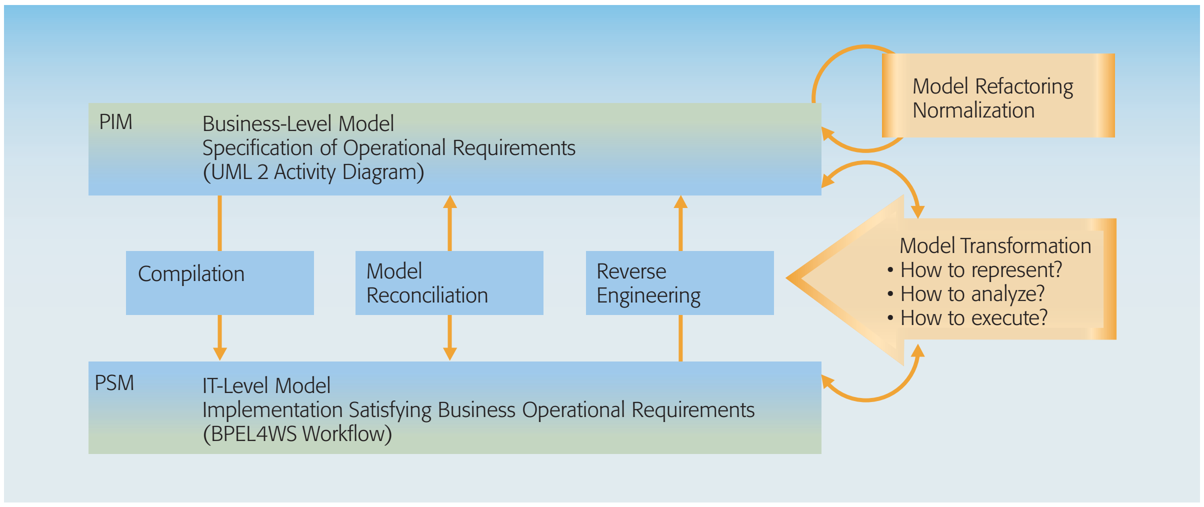

Figure 1
Transformation of PIMs and PSMs

Platform-Independent Models (PIMs) and Platform-Specific Models (PSMs). These models are meant to span the entire life cycle of a software system, facilitate software production and maintenance tasks, and increase software quality.

In order for MDA to succeed as a new and emerging software development paradigm, automatic tools must be available that address two key challenges in particular: the analysis and verification of model-based designs, which allow a human user or an automatic tool to generate new insights into the structural and behavioral features of the modeled system or process; and the mapping and transformation between different models and metamodels with the goal of facilitating conversion to and from PIMs and PSMs. (A metamodel is a precise definition of the constructs and rules needed for creating models.)

Figure 1 shows a set of transformations between a PIM and a PSM. PIMs can be automatically transformed into executable PSMs by compilation techniques. Compilation constitutes the main focus of this paper. We investigate how compilation transforms a PIM (specified by a UML 2.0** [Unified Modeling Language** Version 2] activity diagram[1]) that captures the operational requirements of a process at the business level into an executable PSM—namely, a program in the Business Process Execution Language for Web Service (BPEL4WS)[2] that satisfies the operational requirements captured in the PIM. *Compilation* is a one-way process producing code from a model, where the code and the model may remain linked or evolve independently of each other. In contrast, *model reconciliation* links models together such that they remain synchronized even if those models change, thus requiring bidirectional transformations. Bidirectional transformations between business process models and IT execution models are of particular importance, because they enable organizations to develop business-level and IT-level models at their own pace, while keeping the models synchronized.

One can easily imagine that a compilation transformation could be implemented in some preferred programming language. However, the same questions arise concerning model transformations as those about models themselves; namely, which representation best suits a particular transformation problem, what techniques should be made available to facilitate the analysis and execution of transformations, and so on. A compilation transformation that is represented in a particular programming language can be difficult to analyze and reuse. Furthermore, it is impossible to reverse the compilation program so that it can be used for reverse engineering or model reconciliation.

A *declarative* approach to transformation describes the goal in terms of relations between the initial and final states, and contrasts with an *imperative* approach, which defines explicit intermediate steps to reach the goal. Declarative approaches have a clear advantage in that they can be analyzed, reused, and reversed, and they are investigated in the second part of this paper. This PIM-PSM transformation is presented in the form of declarative rules, whose preconditions and postconditions are described as constraints on models specified in the Object Constraint Language (OCL).[3]

In the future, tools for OCL or other constraint languages will allow us to verify if a transformation rule is executable on a given model or metamodel (i.e., its preconditions can be satisfied), if rules are consistent with each other (i.e., their preconditions and postconditions are not contradictory), and if a rule-based transformation will terminate. So far, no widely agreed-upon approach exists on how to declaratively represent transformations, and the model reconciliation problem has been solved only in a rudimentary manner, namely, under the condition that class diagrams and Java method signatures are synchronized.

This paper is organized as follows. The next section introduces business-process transformation and the business-process diagrams that are the input to our model analysis and transformations. This section also presents an example of a business process, a Web-enabled electronic purchasing system, that will be used throughout the paper. The section following this presents a simple but powerful method to split large and complex business-process diagrams into meaningful subprocesses that are then amenable to automatic compilation into BPEL4WS. The main section introduces an augmented variant of T1-T2[4] analysis as the foundation of our model transformation, followed by a discussion of the technical details of the transformation rules and their declarative encoding in OCL. The paper concludes with a summary and outlook.

BUSINESS PROCESS MODEL TRANSFORMATION

The increasing importance of the model-driven approach for the implementation of business solutions is evident from the involvement of leading IT companies and the ongoing standardization efforts in this area. IBM's WebSphere* Business Integration Modeler Version 5[5] allows designers to automatically generate BPEL4WS programs as well as code for the MQ Series* workflow platform from process models captured in a variant of the Activity Decision Flow Diagram.[6] SAP has partnered with ARIS,[7] IO, and NetWeaver to generate SAP solutions from ARIS models. Microsoft integrates modeling and code generation based on domain-specific languages into its .Net platform.[8] So far, the code generation capabilities of these tools are limited, and they only work for business process models that adhere to numerous restrictions. In particular, flows that contain unstructured cycles or that combine cycles and concurrency cannot automatically be transformed into code. The reasons for these restrictions are twofold:

1. The semantics of business process modeling techniques tends to be on the "intuitive" side, to meet the flexibility needs of business consultants. This can make code generation more difficult, if not impossible, because of semantic ambiguities.[9]

2. Transformation methods that can map large and complex business process models to executable and well-performing code are still under development and are not yet well established in commercial code generation tools.

In our work, we develop transformation methods that target these limitations. In this paper, we report on a novel transformation method that allows us to generate code from graphical business process models containing unstructured sequential cycles. We study subsets of the upcoming Version 2.0 of UML, which focuses in particular on improvements of the modeling of behaviors and processes. UML 2.0 activity diagrams have been greatly enhanced for this purpose and are therefore of major interest as input into our automatic transformation methods. Our target PSM is given by BPEL4WS because of the substantial support this language has in the industry as the upcoming standard for describing Web service orchestrations and because of the role it plays in the WebSphere Business Integration Modeler product.

Business process modeling with UML

Figure 2 shows a model of a business process in the graphical notation of UML 2.0 activity diagrams, which we will use as an example throughout this paper. Because the precise distinction between behaviors, actions, and activities is not yet finalized in UML 2.0, the following terminology is used in this model. The main types of nodes are *action nodes*, which refer to executable actions that may change incoming data, and *control nodes*, which route data

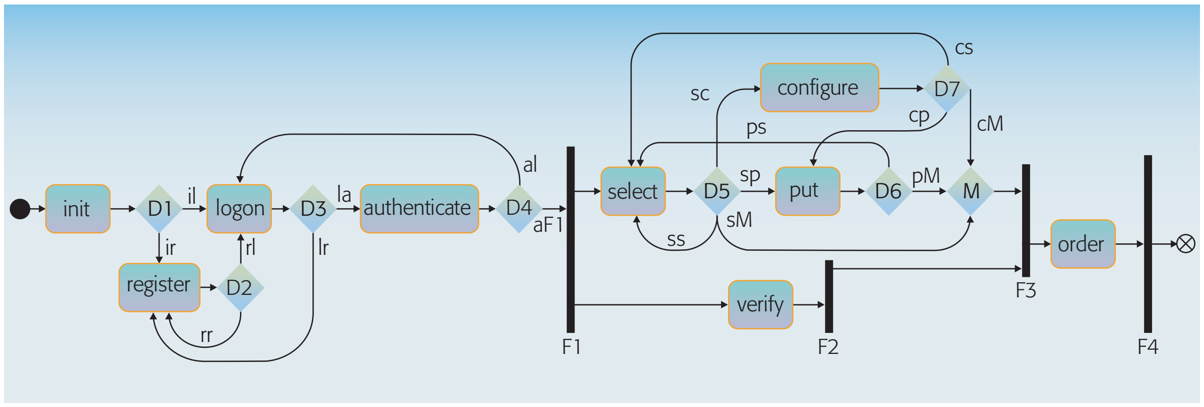

Figure 2
Business-process model of an enterprise customer commerce system

without changing it. In the graphical representation, actions are represented by rounded-corner rectangles. Four types of control nodes are distinguished: *decision nodes* and *merge nodes* are represented using a diamond-shaped symbol, while *forks* and *joins* are represented by vertical bars. In the limited class of activity diagram models that we consider here, edges represent control flow only. The *entry* into a flow is depicted by a solid black circle; whereas, the *exit* from a flow is depicted by a crossed circle. Each edge can be annotated with a guard condition, which must be satisfied in order for the transition to take place.

To describe our example model, we use abstract mnemonic strings to represent these guard conditions—their internal logic is hidden to simplify the following discussion and because it plays no major role in the presentation of the transformation methods. An edge without a guard condition means that the transition is unconditional.

The model depicts the control flow of a Web enterprise customer commerce system (ECCS),[10] which specifies the electronic purchasing of products via a Web interface. The model contains two major business processes: an *authentication process*, in which users register their personal data and log on to the system, and a *shopping process*, in which users select products and submit orders.

The authentication process works on a user-specific authentication data object; whereas, the shopping process works on a data object that combines data relevant to the shopping session with user-specific information. One purpose of the activity diagram is to specify the life cycle of these data objects and the order in which various actions may change them.

The authentication process begins with an *init* (or initialization) step, in which the authentication data object is created for a new user or activated for a user already known to the system. Known users can proceed to *log on* to the purchasing system; otherwise, users are requested to register and submit further information. The *logon* step may fail if the user's registration has expired and all information that was stored has been removed. In this case, the user has to register again. When the logon step succeeds, the user data is verified with an *authenticate* action. All user-relevant data stored in the authentication data object, such as credit card information, open shopping sessions, open and filled orders, as well as delivery addresses, is activated and provided to the subsequent shopping process.

The shopping process contains two concurrent subprocesses, one for *product selection* and one for *verification*. The product selection subprocess allows the user to select and possibly configure products. If a user has an uncompleted shopping session with items placed in the shopping cart but not ordered, the cart is reactivated and the data is made available to the user. New products can be selected and configured until they are finally put into the shopping cart. To keep the process flexible, the user can jump back and forth between the various actions, but the user must leave the product selection process via the *order* action, which handles the submission, cancelation, and deferral of orders placed on products in the shopping cart.

The *verification* subprocess is kept very simple in our example. It contains only a single *verify* action, which stands as a placeholder for any more complex actions or activities that may be needed to ensure the security and integrity of the electronic shopping process. The model specifies only that a *verify* action must have been completed successfully for the *order* action to become executable. Without this information, no order can be submitted, and the process cannot be exited in a valid way. The guard conditions on edges leaving decision nodes are denoted with mnemonic names. For example, the condition for transition from the *init* action to the *logon* action through decision D1 is denoted *il,* while the condition for the transition from the *authenticate* action to fork *F1* through decision D4 is denoted *aF1*.

Semantics of UML activity diagrams

UML 2.0 adopts a token flow semantics to describe when a node in a flow can execute. We assume the following semantics, which closely follows the established interpretation of UML 2.0 activity diagrams:[11]

- An action node can execute when a control token arrives at one of its incoming edges (XOR semantics). It places a control token on its single outgoing edge, which has a guard condition that is always true.
- A decision node can execute when a control token arrives at its single incoming edge. It places a control token on exactly one of its outgoing edges (XOR semantics), namely, the one edge whose guard condition is satisfied. This semantics assumes the guard conditions of all outgoing edges of a decision node are mutually exclusive and exhaustive.
- A merge node has the same token flow behavior as an action node.
- A fork can execute after a token arrives on its single incoming edge. It places control tokens on all of its outgoing edges (AND semantics) and thereby starts parallel executions of flows. We assume that outgoing edges of forks are only annotated with guard conditions that are true.
- A join can execute when control tokens arrive on all of its incoming edges (AND semantics). It places a control token on its single outgoing edge.

Note that we formulated restrictions on the incoming and outgoing edges. These restrictions ensure that our graphical models have nodes with only AND or XOR semantics for the incoming and outgoing edges, but not both, in order to avoid the subtleties of full UML 2.0 activity diagrams. As a consequence, decision nodes in our example can only start a single flow, and merge nodes always have only one active flow arriving. Concurrent flows can only be triggered by forks, but not by actions or decision nodes as is possible in the full UML 2.0 activity diagrams.

Model transformation of full UML 2.0 activity diagrams requires the formalization of their semantics with greater precision than is available today. Preliminary approaches to formalizing UML subsets based on Petri nets,[12] labeled transition systems,[13] abstract state machines,[14] and Pi-calculus[15] have been presented recently. The semantics are usually used to devise techniques to automatically verify properties of the activity diagrams. However, none of these formalizations is worked out in such detail that it can help in developing sound transformation, normalization, or refactoring algorithms that transform one arbitrary UML 2.0 activity diagram into another, equivalent representation. Similarly, formalizations of BPEL4WS have been developed based on communicating automata,[16] abstract state machines,[14] and Petri nets,[17] but scalable algorithms that allow a tool to automatically reason about a given BPEL4WS program are only in the initial state of development.[16,18]

AUTOMATIC ANALYSIS OF CONTROL-FLOW MODELS

The informal description of the ECCS example clearly distinguished several meaningful subprocesses, which interact with each other to implement the ECCS system. The control-flow model for this example is "flat" in the sense that it does not use structured activities to create a hierarchical model in which the various subprocesses can be easily identified. In order to improve the scalability and performance of model compilation techniques, it makes sense to complement them with analysis techniques that can identify such subprocesses instead of relying on the human designer to provide a well-structured hierarchical model. Today's workflow code-generation tools lack such analysis capabilities and would map large, flat models to large, monolithic code.

The close relation of control-flow models to control-flow graphs of programming languages enables the use of techniques from the fields of compiler optimization and program synthesis in addressing the model transformation problem. Control-flow analysis can reveal very interesting insights into a structure of a business process that can later be exploited to generate optimized workflow instructions.

In the following, we discuss two main techniques: an automatic analysis to discover subprocesses in large, flat control-flow diagrams based on so-called two-terminal regions, and an automatic analysis of how to transform sequential flows containing unstructured cycles into compact workflow code based on the notion of reducibility from compiler theory. In the latter technique, flows with unstructured cycles that have a nonreducible control-flow graph are transformed into a state machine represented in BPEL4WS, while flows with unstructured cycles that have a reducible control-flow graph are normalized and encoded in BPEL4WS.

Identification of subprocesses

A closer look at Figure 2 shows that structural regions in the control-flow graph correspond to the ECCS subprocesses discussed previously. These regions are characterized by sets of nodes that have a single entry node to the region and a single exit node from the region. This structural characteristic seems to be quite common in control-flow models, and it can be exploited to identify regions in the process model that can be analyzed independently of each other. Flow-graph analysis of computer programs has defined so-called two-terminal regions,[19] which provide an appropriate definition of the structures in which we are interested. We next review some relevant concepts from compiler theory[20] that form the foundation for our analysis and transformation techniques.

Definition 1. In a directed graph with one entry and one exit node, a node n is said to *dominate* (or *predominate*) a node m if every path from the start node to node m goes through node n. The dominance relation is transitive; that is, if a node n_1 dominates a node n_2 and n_2 in turn dominates another node n_3, then n_1 also dominates n_3.
Definition 2. A node m *postdominates* a node n if every path from node n to the exit node goes through node m. If there is an edge from a node n to a node m in the graph, then n is called the *predecessor* of m, while m is called the *successor* of n.
Definition 3. A *two-terminal region* is a subgraph where

1. a single entry node exists in the set that dominates all other nodes in the set, as well as all their predecessors, which must also be in the set, and
2. a single exit node exists in the set that postdominates all other nodes in the set as well as their successors, which must also be in the set.

The two requirements in this definition ensure that (1) there is no path from the outside into the two-terminal region that does not pass through the entry node, and (2) there is no path from the two-terminal region to the outside that does not pass through the exit node.

The notion of two-terminal regions can be used as a heuristic to discover subprocesses in large control-flow graphs. Applied to the ECCS example, the heuristic identifies the following regions:

1. A region R1 comprising the *init* (entry node), *register*, *logon*, and *authenticate* actions and the decision nodes D1, D2, D3, and D4 (the exit node).
2. A region R2 with entry node fork F1 and exit node join F4 to comprise the *verify, select, configure, put,* and *order* actions.
3. A nested region R3 inside region R2 with entry node *select,* exit node merge M, and the *configure* and *put* actions.

After having identified these three possible regions in the ECCS, each of the regions can be analyzed separately. Regions R1 and R3 comprise unstructured cyclic flows, which are sequential. No parallelism can occur because of the restrictions we imposed on the control-flow semantics. Unstructured cycles have more than one entry or exit point into or out of the cycle. Region R2 is a concurrent region, in which the cyclic *product-selection* process synchronizes with the simple *verification process,* but the synchronization link lies outside the unstructured cycle. The cycle contains the *select, configure,* and *put* actions; whereas, the synchronization is established between the *verify* and *order* actions.

The structure of cycles and the types of synchronization (between cyclic or linear threads) have to be transformed into functionally equivalent BPEL4WS code. Functional equivalence establishes a desirable correctness criterion for our subsequent transformation. When provided with the same input data, the original control-flow model and the generated, functionally equivalent BPEL4WS model yield the same output. Functionally equivalent transformations of Turing-complete programs are always possible,[21] but may require the introduction of new guard variables and the duplication of code in the case of *nonreducible flow graphs*. In the worst case, the transformation may lead to an exponential expansion (or "blow-up") of the program.[20,22]

Transforming the unstructured cyclic flows contained in regions R1 and R3 of the ECCS to BPEL4WS poses a particular problem, because the BPEL4WS language forbids unstructured cycles. Link elements in BPEL4WS define synchronization, and each link element has a source and a target activity. Control flow in BPEL4WS can be specified by using explicit BPEL4WS *link* elements or BPEL4WS structured activities such as *sequence, switch,* and *while,* the latter only allowing the designer to describe structured cycles. The *while* activity has a single exit, and it is exited when its defined termination condition is no longer valid. No unstructured control flow into or out of a *while* activity is permitted because *link* elements are not allowed to have their source activity within a *while* activity and their target activity outside, or vice versa. Furthermore, the control flow defined via *link* elements must be acyclic, that is, no cycles can be established via links.

Unstructured cyclic flows can be transformed into well-formed BPEL4WS by adapting control-flow normalization techniques from compiler theory.[23,24] However, these techniques are currently limited to nonconcurrent cycles only. Fortunately, regions R1 and R3 are nonconcurrent and can therefore be transformed using these techniques. The concurrent region R2 is acyclic when we consider R3 as a single structured activity node. Furthermore, synchronization only takes place between sequential actions that are not part of any loop. ***Figure 3*** outlines the structure of the BPEL4WS code which results from our analysis.

The ECCS model can be considered as a typical example that falls into a limited, yet practically relevant class of concurrent processes, which seems to occur frequently in business process designs. In this class, each of the parallel flows may contain unstructured cycles, but any synchronization between the parallel flows only takes place between BPEL4WS activities lying outside of cycles. In the next subsection, we describe the basic principles underlying the transformation of regions R1 and R3 into functionally equivalent BPEL4WS code.

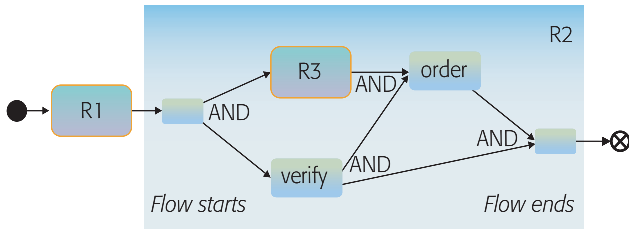

Figure 3
Outline of the generated BPEL4WS code

Reducibility of identified regions

As we briefly mentioned previously, the transformation of unstructured cycles may lead to an exponential expansion of the transformed program. This problem occurs in the case of *nonreducible flow graphs*. The notion of *reducibility* plays a major role in compiler theory.[20] In fact, many normalization and optimization techniques apply to reducible control-flow graphs only. Several techniques to determine the reducibility of control-flow graphs exist, one of the most established techniques being T1-T2 analysis.[4] In this technique, T1 and T2 are rules which stipulate the following:[20]

Rule T1. If n is a node with a loop, that is, it has an edge which starts and ends at node n, delete that edge.

Rule T2. If there is a node n (not the initial node) that has a unique predecessor m, then m may consume n by deleting n and making all successors of n (including, possibly, m) successors of m.

After rules T1 and T2 have been applied (in any order), the *limit-flow graph* can be computed. The limit-flow graph remains when the rules T1 and T2 have been applied repeatedly until neither rule is applicable. If the limit-flow graph collapses into a single node, then the flow graph is reducible;

otherwise, it is not. *Figure 4* shows the results of the T1-T2 analysis of region R1, wherein three nodes remain, all of which lack a unique predecessor. Consequently, R1 is nonreducible. We omit the analysis of R3, which collapses into a single node and thus is reducible.

As a result of this analysis, we know that any normalization of the control flow of R1 which replaces unstructured cycles with structured cycles and possibly additional guard conditions will require the duplication of action nodes. We also know this duplication can be exponential in the worst case. This is not very desirable when we think of executing such a flow in BPEL4WS code. A better alternative is to proceed with normalization until the limit-flow graph is reached and then map the control flow encoded in the limit-flow graph directly to a state machine encoded in BPEL4WS. Region R3 can be normalized without the danger of code duplication because its underlying control-flow graph is reducible. We discuss both transformations in the next section.

TRANSFORMATION OF CYCLIC SEQUENTIAL PROCESSES INTO BPEL4WS

This transformation is based on augmented T1-T2 rules, which collapse a control flow into a graph with fewer top-level nodes and edges, following which BPEL4WS code can be generated. T2 takes a node *n* and its predecessor *m* in the activity diagram and merges them into a new node *s*. T1 takes a node *n* in the activity diagram, removes the self edge, and turns it into a loop node. If the original activity diagram was reducible, the transformation terminates with the activity diagram reduced to a single top-level structured activity node, at which point it can then be transformed trivially to a BPEL4WS model. In the case of nonreducibility, several structured activity nodes remain after T1 and T2 have been applied. In this case, our approach is to transform the resulting activity diagram to a BPEL4WS model by mapping the remaining nodes to a state machine encoded in BPEL4WS. The transformations (described in the following subsection) are applied only to regions in the process model which comprise sequential processes without concurrency.

Transformation of nonreducible cyclic flows

The state machine that is used in the case of nonreducibility is a behavioral state machine in the sense of UML 2.0. It is a convenient representation to define the life cycle of objects—an idea also proposed by the approach described in *ADoc-oriented programming.*[25,26] In our case, the life cycle of authentication data consisting of identities, passwords, and other user-specific data is described with this state machine.

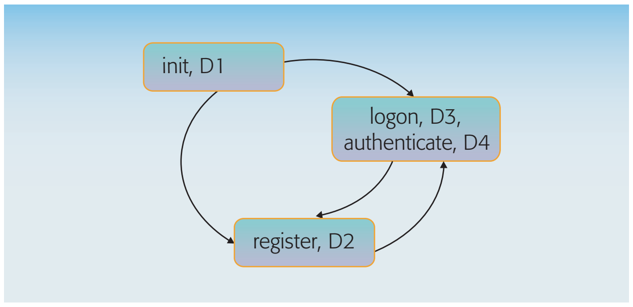

Figure 4
The nonreducible limit-flow graph of region R1

The control flow diagram of region R1 is transformed into a state machine in a straightforward way by using the T1 and T2 rules until a limit-flow graph is reached that is not reducible further. *Figure 5* illustrates the results of this transformation. Rule T2 merges all decision nodes with their preceding actions and accordingly updates the outgoing control edges exiting the actions. It also merges the *logon* and *authenticate* actions into a new structured activity node. This new node, as well as the *register* action, has a self loop. To both loops, rule T1 can be applied, introducing two new loop nodes, which we represent in textual form on the right side of Figure 5. With these transformation steps, the limit-flow graph is reached, and none of the rules is applicable anymore.

The remaining structured activity nodes are now mapped to the BPEL4WS model encoding the generated state machine within a BPEL4WS *while* activity. The *init* action can be placed outside the *while* activity because none of the other actions has an outgoing edge leading back to *init*. In the BPEL4WS model, the iteration of the *while* activity is controlled by the value of a *next* variable that is initialized after the start action *init* is invoked. Each action is encoded as a Web service invocation. These BPEL4WS *invoke* activities are synchronous, resulting in the guard variables immediately being updated. We omit the details concerning how the

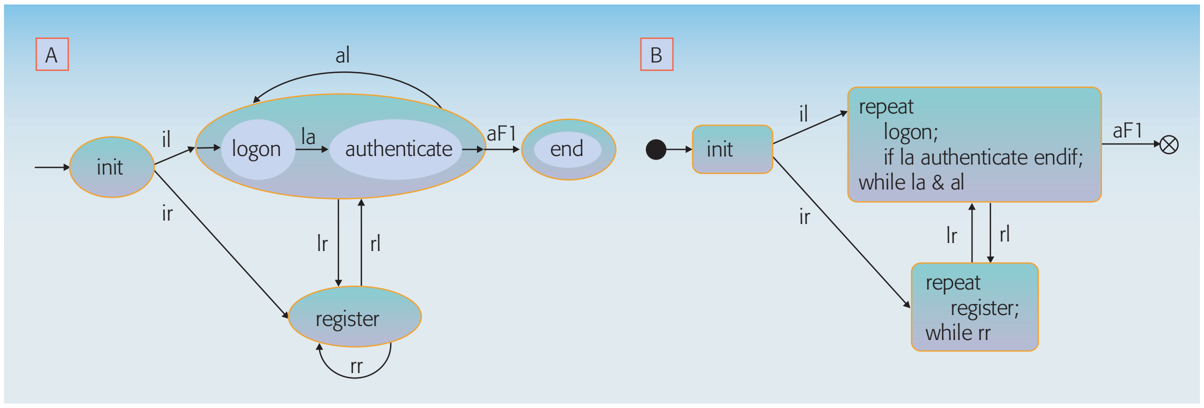

Figure 5
Results of T1-T2 transformation; (A) the state machine controller for the authentication subprocess; (B) the partially reduced UML 2.0 activity model upon termination of T1-T2 analysis

guards are encoded, but one can imagine that a guard condition may test the values of the authentication data represented in a WSDL (Web Services Description Language) message that is manipulated by the activities. Only when the condition *aF1* in the state *authenticate* is true, is the value *end* assigned to the *next* variable and the final state reached. The resulting BPEL4WS code is shown in ***Figure 6***.

In Reference 27, a representation of state machines in the form of "ADocs" with a mapping to J2EE**-specific runtime components like session and entity beans is discussed. In our case, we encode state machines in BPEL4WS code, and this is their executable model. The advantage of the encoding is that we can easily combine state machines with Petri-net-like flows by using the same runtime model. Furthermore, BPEL4WS provides a straightforward means of communication between state machines and flows by means of Web services.

Transformation of reducible cyclic flows

The transformation of reducible cyclic flows proceeds in the same way as the transformation of nonreducible cyclic flows, the only difference being that the activity diagram can be reduced to a single structured activity node containing appropriate loop activities instead of unstructured cycles. Thereafter, a BPEL4WS model is generated by introducing the corresponding *sequence*, *switch*, and *while* activities. ***Figure 7*** summarizes the transformation steps performed on region R3. Edges to which a transformation is applied are shown with dashed lines.

The result of the transformation is a mapping between the activity nodes in the activity diagram and the structured activity nodes in the reduced activity diagram.

Figure 7 depicts a specific order of the application of the two rules. For example, in step B, *merge* and *put* could alternatively be transformed by using T2, while in step C, T1 could be applied to remove the self edge. From the point of view of the activity diagram, any order of rule application will result in a single activity node in the final model. From the point of view of the BPEL4WS model generated after the reduction steps, the order in which the rules are applied determines the structure of the BPEL4WS code. By choosing a specific order in the reduction steps, BPEL4WS code can be optimized with respect to various requirements—an issue further discussed in Reference 27.

In the following section, we discuss the declarative representation of the T1-T2 rules in OCL and briefly describe the mapping from T1-T2 reduced activity diagrams to BPEL4WS models.

DECLARATIVE REPRESENTATION OF AUGMENTED T1-T2 RULES

It is quite obvious how the transformation rules T1 and T2 can be implemented in a procedural manner. However, we are interested in investigating whether a declarative representation of these rules is possible and what its advantages and disadvantages are. We are motivated by the ongoing discussions around

```
<process>
   <sequence>
      <invoke init/>
      <switch>
         <case condition= 'il'>
            <assign 'next:=logon'/>
         </case>
         <case condition= 'ir'>
            <assign 'next:=register'/>
         </case>
      </switch>
      <while condition= 'next!=end'>
         <switch>
            <case condition= 'next=logon'>
               <assign 'looping_l:=true'/>
               <while condition= 'looping_l'>
                  <invoke logon/>
                  <switch>
                     <case condition= 'la'>
                        <invoke authenticate/>
                     </case>
                  </switch>
                  <assign 'looping_l:=la & al'/>
               </while>
               <switch>
                  <case condition= 'lr'>
                     <assign next:=register/>
                  </case>
                  <case condition= 'la & aF1'>
                     <assign next:=end/>
                  </case>
               </switch>
            </case>
            <case condition= 'next=register'>
               <assign 'looping_r:=true'/>
               <while condition= 'looping_r'>
                  <invoke register/>
                  <assign 'looping_r:=rr'/>
               </while>
               <switch>
                  <case condition= 'rl'>
                     <assign next:=logon/>
                  </case>
               </switch>
            </case>
         </switch>
      </while>
   </sequence>
</process>
```

Figure 6
Generated abstract BPEL4WS code for region R1

the emerging OMG standard for Query/Views/Transformations (QVT).[28] Many submissions in response to the QVT Request for Proposals (RFP) envision the use of declarative or hybrid languages to represent queries, views, and transformations on models. Among the various language proposals, OCL is a favorite.[3,29]

Role of OCL in model transformation

OCL is the standard language in UML for specifying constraints on object-oriented models. UML Version 1 uses OCL either to add invariants to a model or parts of it or to specify the preconditions and postconditions for dynamic UML elements such as state machines or methods. In UML 2.0, OCL is no longer limited to expressing constraints of models, but is envisioned as a general-purpose language to express queries, transformations, arbitrary conditions, and business rules.[3] The use of OCL for model transformations is also supported by the revised QVT submission,[30] where OCL is extended to describe relations between models. In our case study, we used OCL to define the preconditions and postconditions for our augmented T1-T2 rules. The precondition captures necessary and sufficient conditions that determine when a rule is applicable. The postcondition describes the intended update to the model, that is, the effect of the transformation.

An OCL specification consists of two parts, a context and a set of expressions. For example, we show the `dequeue` method that returns the first element of a queue and removes that element from a queue:

```
context Queue::dequeue(): QueueElement
pre: self.notEmpty
post: self.size = self.size@pre - 1
post: result = self.firstElement@pre
```

The precondition ensures that the queue is not empty (`self` refers to an object of type Queue). The postcondition states that the queue, when the method terminates, will be shorter by one element, and the first element of the original queue will be returned. The OCL operator `@pre` is used in the postcondition to reference the value of a model element at the beginning of the computation; for example, `q.size@pre` denotes the value of `q.size` when the method begins to execute. The OCL reserved word `result` denotes the result that is computed by a method.

OCL allows the modeler to navigate within an object-oriented model. ***Figure 8*** shows a fragment of the UML 2.0 activity diagram metamodel, on which the transformation operates. An *ActivityNode* in the model has an association with a set of edges named `incoming`. Thus, `N.incoming` denotes the set of edges that enter node *N*. Operations on sets are performed using the operator → as, for

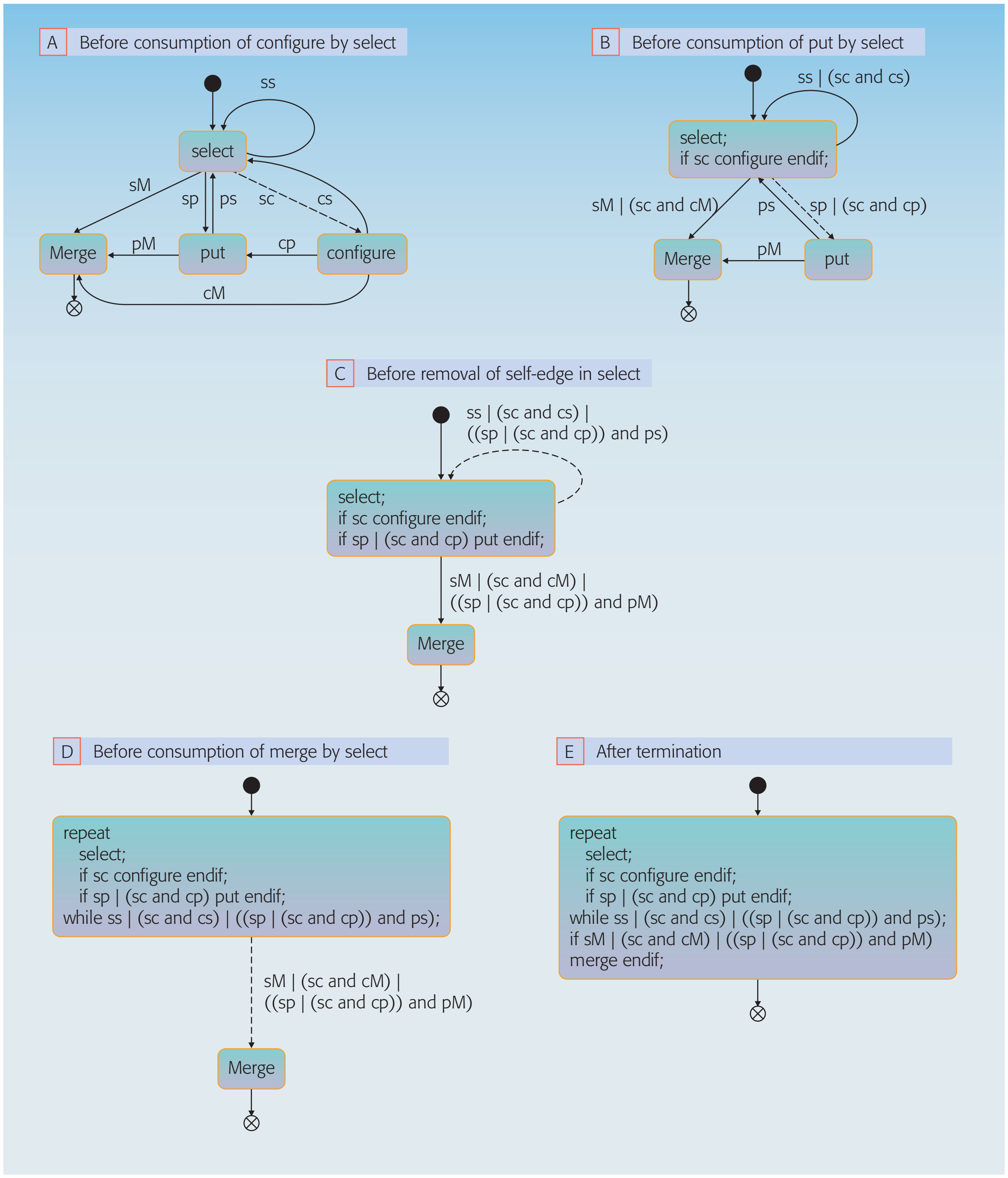
A Before consumption of configure by select
ss
select
sM
sp
ps
sc
cs
Merge
pM
put
cp
configure
cM
B Before consumption of put by select
ss | (sc and cs)
select;
if sc configure endif;
sM | (sc and cM)
ps
sp | (sc and cp)
Merge
pM
put
C Before removal of self-edge in select
ss | (sc and cs) |
((sp | (sc and cp)) and ps)
select;
if sc configure endif;
if sp | (sc and cp) put endif;
sM | (sc and cM) |
((sp | (sc and cp)) and pM)
Merge
D Before consumption of merge by select
repeat
select;
if sc configure endif;
if sp | (sc and cp) put endif;
while ss | (sc and cs) | ((sp | (sc and cp)) and ps);
sM | (sc and cM) |
((sp | (sc and cp)) and pM)
Merge
E After termination
repeat
select;
if sc configure endif;
if sp | (sc and cp) put endif;
while ss | (sc and cs) | ((sp | (sc and cp)) and ps);
if sM | (sc and cM) | ((sp | (sc and cp)) and pM)
merge endif;

Figure 7
Control-flow normalization of region R3

example, in `N.incoming→isEmpty()`. Navigations can be chained by using the OCL shorthand notation for the *collect* operation; for example, `N.incoming.source` denotes the set of nodes from which an edge leads to *N*.

OCL provides different types of collections and a considerable set of operations. We use the type `Set`, which corresponds to the mathematical definition of a set (i.e., elements are not ordered, and each element occurs at most once) and the type

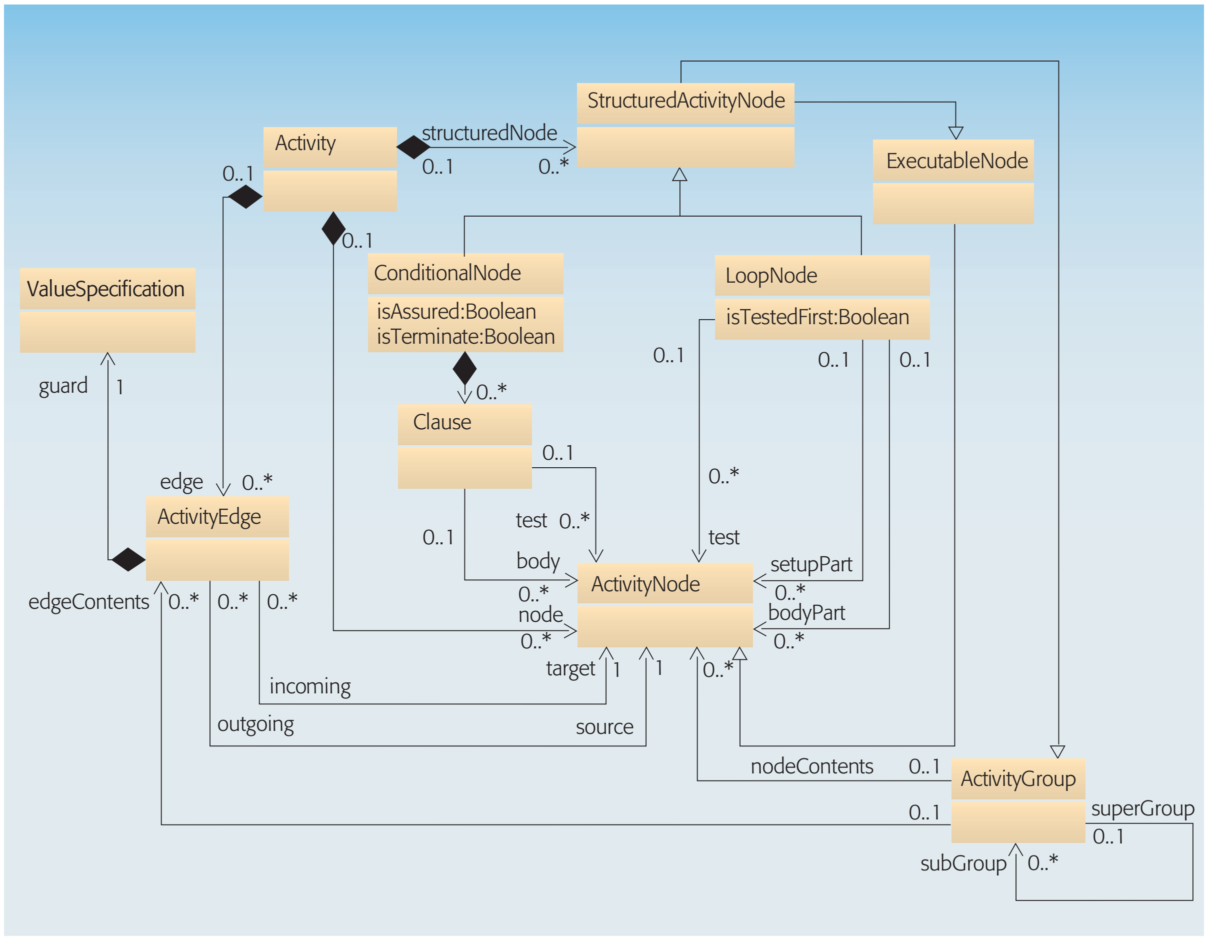

Figure 8
Fragment of the UML 2.0 activity diagram metamodel

`Sequence`, which corresponds to a list (i.e., elements are ordered and may occur more than once). Examples of collections are `Set{1,2,3}` and `Set{e}`. The former denotes a set containing the numbers 1, 2 and 3. The latter denotes a set that contains a single element *e*.

As OCL expressions tend to become rather lengthy, `let` expressions can be used to introduce new variables that abbreviate expressions. For example,

```
let (nodes = N.incoming.source) in
  nodes→notEmpty()
```

defines `nodes` as an abbreviation for `N.incoming.source`, which can then be referenced in the OCL expression `nodes→notEmpty()`. The scope of `let` is limited to a single OCL expression.

To summarize, OCL provides features to navigate within object-oriented models, to deal with model elements, and to express statements about the result of an operation and the state of the model before an operation is executed. This justifies the emphasis given to OCL by the upcoming QVT standard.

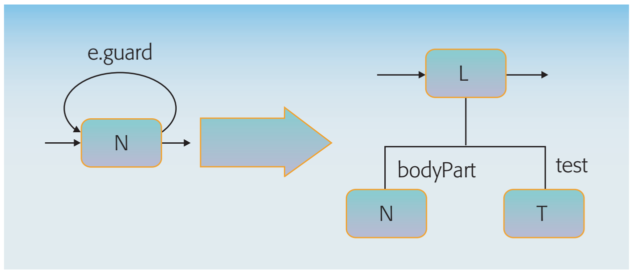

Figure 9
T1 rule augmented by generation of loop activities

```
context ActivityDiagram: :augmented-T1(N:ActivityNode, e:ActivityEdge)

pre: N.incoming->intersect(N.outgoing) = Set{e}

post: LoopNode.allInstances()->exists(L |                          -- line  1
          L.oclIsNew() and                                         -- line  2
          L.incoming = N.incoming@pre->excluding(e) and            -- line  3
          L.outgoing = N.outgoing@pre->excluding(e) and            -- line  4
          L.bodyPart = N and                                       -- line  5
          L.isTestedFirst = false and                              -- line  6
          ActivityNode.allInstances->exists(T |                    -- line  7
            T.oclIsNew() and                                       -- line  8
            L.test = T and                                         -- line  9
            node2guardEquivalence(T, e.guard@pre)                  -- line 10
           ) and                                                   -- line 11
          N.incoming->isEmpty() and                                -- line 12
          N.outgoing->isEmpty()                                    -- line 13
        )
```

Figure 10
OCL constraint for rule T1

Representing the augmented T1 rule in OCL

Figure 9 shows the desired transformation: a node *N* with a self-looping edge *e* that is guarded by *e.guard* is transformed into a set of three nodes *L*, *T*, and *N* that form the head, test expression, and body of the loop expression. In earlier figures, for the purposes of aiding comprehension, we depicted the loop node and its accompanying elements as text embedded in the corresponding activity node. In Figure 9, we show instead a view of the transformation in terms of the metamodel instances that are involved in the transformation. This approach is taken to clarify the meaning of the OCL specifications.

The OCL constraint shown in *Figure 10* defines the augmented T1 rule for any activity node *N* that is passed as a first argument to the transformation function. The second parameter, of type *ActivityEdge*, references the edge *e* which forms the self-loop. The relationship between the activity node *N* and activity edge *e* is asserted by the precondition. The transformation demands the creation of an object *L* of type *LoopNode* that embeds the former node *N* as its body and a node *T* for the evaluation of the loop's test expression.

The postcondition describes the update in the activity diagram: In lines 1 and 2, we postulate the existence of an object *L* as an instance of the *LoopNode* type. Lines 3 to 5 postulate that *L* replaces *N*, becoming the top-level activity node with respect to the given activity nodes. As the body of a node with a self-loop is executed at least once, the property *isTestedFirst* is set to *false* (line 6) to ensure the same semantics. Lines 7 to 11 demand the existence of a set of nodes that compute a Boolean value to determine if another execution of the body will be performed. Note that the function *node2guardEquivalence* in line 10 is a predicate that is used to define the equivalence between the guard condition of the *ActivityEdge e* and the loop condition *T* for the *LoopNode L*. Finally, lines 12 and 13 ensure that node *N* will not have any incoming or outgoing edge after the transformation.

Representing the augmented T2 rule in OCL

The transformation performed by T2 in the activity diagram is much more complicated than is the case for rule T1. Similarly to the previous subsection, we describe the transformation in terms of the meta-model instances that are involved, as depicted in *Figure 11*. A *ConditionalNode C* and a *StructuredActivityNode S* are introduced with the transformation. It places *C* and *ActivityNode M* into *S*, and it takes the guard condition of edge *e* as the test condition of *C* and the body of node *N* as the body of *C*. Furthermore, transforming the outgoing edges of the involved nodes requires the identification of distinct subsets of outgoing edges and the merging of certain subsets among them.

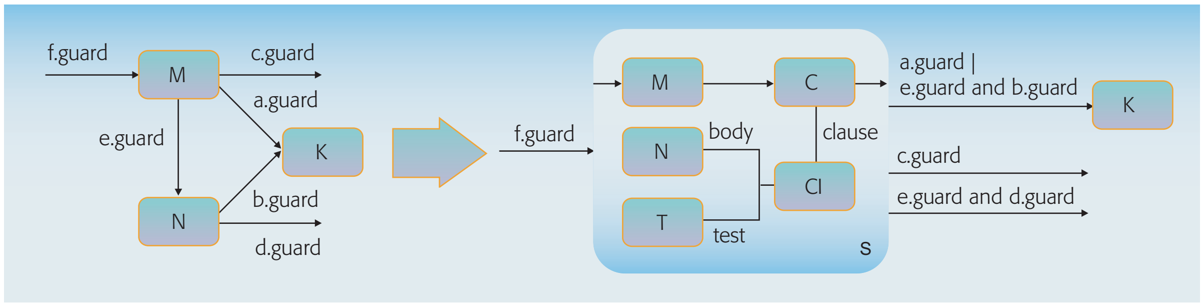

Figure 11
T2 rule augmented by generation of *ConditionalNode* activities

In the specification for the `augmented-T2` rule shown next, the parameters *M* and *N* denote the involved nodes, and *e* represents the guarded edge between them.

```
context ActivityDiagram::augmented-T2
  (M: ActivityNode, N: ActivityNode,
     e:ActivityEdge)
```

The precondition asserts that *M* is the only predecessor of *N*, *e* is the activity edge connecting them, and *e* is the only incoming edge of *N*.

```
pre: M.outgoing→includes(e) and
     N.incoming = Set{e}
```

The postcondition specifies the merging of *N* and *M* in a new activity node *S*. In ***Figure 12***, lines 1 and 2 specify the creation of the new *StructuredActivity-Node S*. Line 3 asserts that *S* takes the incoming edges of *M*. (Note that changes to the outgoing edges are postulated in lines 27–54 and are explained below). Lines 4 and 5 assert that a new *ConditionalNode C* is created. Line 6 asserts that *M* and *C* are subnodes of *S*. Lines 7 and 8 assert that a new *ActivityEdge m2ce* is created. Lines 9–11 assert that *m2ce* connects *M* to *C*, and it is contained in *S*. Lines 13–21 assert that a newly created *Clause Cl* has *N* as its body and a newly created *ActivityNode T*, which has an equivalent condition to *e.guard*, as its test condition. Lines 22–25 assert that *M* and *N* are no longer connected by edges, except for a single edge to *C* for *M*.

Furthermore, the postcondition has to specify the update on the outgoing edges. Two edges that join the same source and target nodes can be merged into a single edge whose guard condition is the disjunction of the original guards, i.e.,

$X \xrightarrow{e_1} Y, X \xrightarrow{e_2} Y$ becomes $X \xrightarrow{e_1 \vee e_2} Y$.

Similarly,

$X \xrightarrow{e_1} Y, Y \xrightarrow{e_2} Z$ becomes $X \xrightarrow{e_1 \wedge e_2} Z$.

Three different sets of edges are distinguished for the update of the outgoing edges that leave nodes *M* and *N*, depicted in Figure 11. Each of the sets is captured by a new definition that we introduce via a `let` expression within the postcondition.

The set of edges containing the example edge guarded by *c.guard* leaves *M* but does not enter any node to which outgoing edges from *N* lead as well. Furthermore, edge *e* that connected *M* and *N* must be removed. We capture this subset in the variable `set1` (Figure 12, lines 27–29).

The set containing the edge guarded by *d.guard* captures the outgoing edges of node *N*. These edges have to be added to node *C* as additional outgoing edges. Their guard conditions have to be conjunctively joined with the guard condition *e.guard* because *N* was reachable from *M* via *e* only. We capture this subset in the variable `set2` (Figure 12, lines 30–38).

The subset containing the example edges guarded by *a.guard* and *b.guard* captures the outgoing edges of nodes *N* and *M*, which enter the same target node *K*. The new node *S* inherits these two edges. However, they can be combined into a single edge because of the common target node. The guard condition of the new resulting edge leaving *S* is the disjunction of the conditions *a.guard* and *b.guard*, where *b.guard* is conjunctively joined with the guard of *e*. We capture this subset in the variable

```
post: StructuredActivityNode.allInstances()->exists(S |                        -- line  1
          S.oclIsNew() and                                                     -- line  2
          S.incoming = M.incoming@pre and                                      -- line  3
          ConditionalNode.allInstances()->exists(C |                           -- line  4
              C.oclIsNew() and                                                 -- line  5
              S.nodeContents = Set{M, C} and                                   -- line  6
              ActivityEdge.allInstances()->exists(m2cE |                       -- line  7
                 m2cE.oclIsNew() and                                           -- line  8
                 m2cE.source = M and                                           -- line  9
                 m2cE.target = C and                                           -- line 10
                 S.edgeContents = Set{m2cE}                                    -- line 11
              ) and                                                            -- line 12
              Clause.allInstances()->exists(Cl |                               -- line 13
                 Cl.oclIsNew() and                                             -- line 14
                 C.clause = Cl and                                             -- line 15
                 Cl.body = N and                                               -- line 16
                 ActivityNode.allInstances()->exists(T |                       -- line 17
                    Cl.oclIsNew() and                                          -- line 18
                    Cl.test = T and                                            -- line 19
                    node2guardEquivalence(T, e.guard@pre)                      -- line 20
              ))    and                                                        -- line 21
              M.incoming->isEmpty() and                                        -- line 22
              M.outgoing.target = Set{C} and                                   -- line 23
              N.incoming->isEmpty() and                                        -- line 24
              N.outgoing->isEmpty()                                            -- line 25
          )  and                                                               -- line 26
          let    set1: Set(ActivityEdge) = M.outgoing@pre->select(x |          -- line 27
                    N.outgoing@pre.target@pre->excludes(x.target)              -- line 28
                 )->excluding(e),                                              -- line 29
                 set2: Set(ActivityEdge) = N.outgoing@pre->select(x |          -- line 30
                    M.outgoing@pre.target@pre->excludes(x.target))->iterate(   -- line 31
                       d: ActivityEdge, resultSet: Set(ActivityEdge) = Set{} | -- line 32
                          resultSet->including(ActivityEdge.allInstances()->any(y |  -- line 33
                             y.oclIsNew() and                                  -- line 34
                             y.source = e.source and                           -- line 35
                             y.target = d.target and                           -- line 36
                             y.guard = e.guard.and(d.guard)                    -- line 37
                 ))),                                                          -- line 38
                 set3: Set(ActivityEdge) = M.outgoing@pre->select(x |          -- line 39
                    N.outgoing@pre.target@pre->includes(x.target))->iterate(   -- line 40
                       a: ActivityEdge, resultSet: Set(ActivityEdge) = Set{} | -- line 41
                         ActivityEdge.allInstances()->exists(y,z |             -- line 42
                             y.oclIsNew() and                                  -- line 43
                             y.source = a.source and                           -- line 44
                             y.target = a.target and                           -- line 45
                             y.guard = a.guard.or(z.guard) and                 -- line 46
                             resultSet->including(y) and                       -- line 47
                             z.oclIsNew() and                                  -- line 48
                             z.source = e.source and                           -- line 49
                             let b = N.outgoing@pre->any(w | w.target=a.target) in  -- line 50
                                z.target = b.target and                        -- line 51
                                z.guard = e.guard.and(b.guard)                 -- line 52
                 )) in                                                         -- line 53
         S.outgoing = set1->union(set2)->union(set3)                           -- line 54
     )                                                                         -- line 55
```

Figure 12
OCL code of postcondition for rule T2

`set3` (Figure 12, lines 39–53), and the update of the set of outgoing edges of *C* can be conveniently expressed as the union of the three sets, `set1`, `set2`, and `set3` (Figure 12, line 54).

Structured activity diagram to BPEL4WS

The transformation of T1-T2 normalized activity diagrams to BPEL4WS models can also be described in terms of OCL preconditions and postconditions.

As the transformation is relatively straightforward compared to the T1 and T2 reduction steps, we do not present the corresponding OCL conditions here.

Once reduced with T1 and T2, an activity diagram consists of *ConditionalNodes* and *LoopNodes* at the top level, which are ordered by *ActivityGroups*. In BPEL4WS, a *sequence* is used to establish an ordering on a set of activities. As such, the transformation of the normalized activity diagram involves mapping *ConditionalNodes* to BPEL4WS *switch* elements and *LoopNodes* to *while* elements, taking into account the different semantics of the corresponding constructs.

The transformation of a conditional construct from activity diagrams to BPEL4WS code is almost trivial, as there are one-to-one correspondences between the element pairs *ConditionalNode* and *switch*, *Clause* and *case*, and *body* and *activity*. Again, we postulate the existence of a function to convert the guard condition, as with *node2guardEquivalence* as mentioned previously.

The transformation of a *LoopNode* in UML 2.0 to a *while* node in BPEL4WS raises a problem, as these constructs have one important semantic difference: if the property *isTestedFirst* is false (and it will always be false in our transformation environment, as seen in line 6 of Figure 10), the body of a *LoopNode* is executed once before the loop condition is tested. In BPEL4WS however, the loop condition is always tested first when the control flow reaches a *while* node. As described in Reference 23, a specific variable, *looping*, may be introduced to store the value of the termination condition. It is set to true before the *while* is executed for the first time, and this ensures that the loop body is executed at least once. Afterwards, *looping* is assigned the value of the original loop condition in the body of the *while*, so the original activity *N* changes to *N′*. ***Figure 13*** shows the structure of the translation; the code regarding the variable *looping* is illustrated in ***Figure 14***.

Evaluation of OCL for model transformation

OCL provides a precise, formal, and typed language to specify the preconditions and postconditions of a transformation rule. In the traditional sense, such an OCL specification can be used to verify the correctness of an implementation and as the basis for an implementation. There are a number of examples of OCL being used for describing transformations, for example, References 31 and 32. Apart from its sometimes verbose form, OCL is well-suited for describing transformation rules on models, mainly due to its tight integration with object-oriented models or metamodels.

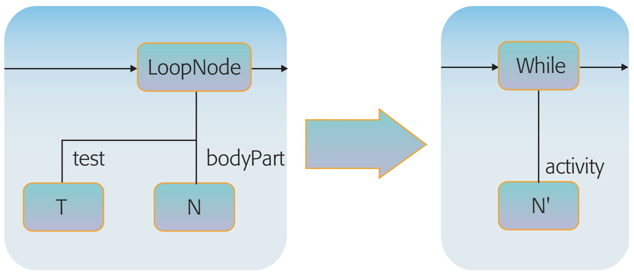

Figure 13
Transformation of a LoopNode to a corresponding *while* activity

In a new direction, precondition and postcondition pairs can be given an execution semantics; this means that as long as they are internally consistent, that is, there are no contradictions in either condition and their termination is ensured, a precondition and postcondition pair can be executed to realize the effect described by the semantics. The precondition is interpreted as the means for determining the applicability of a transformation rule, that is, a necessary and sufficient condition for execution. The postcondition is interpreted as a goal statement that is to be realized when the precondition is satisfied. Warmer et al. propose this kind of framework for OCL in References 3 and 33, and the framework is further developed in Reference 34. They extend the notion of precondition and postcondition by introducing symmetry in the OCL specification. More precisely, they replace a precondition and postcondition pair with two patterns. These patterns are described as conditions in OCL, and they are used to determine the applicability of the transformation rule, one for each direction of the transformation (i.e., activity diagrams to BPEL4WS and vice versa). To describe the transformation, an additional pattern is used to define the correspondences between elements in the source model and the target model.

The framework of Warmer et al. greatly improves the ease with which transformations can be described in OCL. However, there are some issues

```
<process>
  <sequence>
    <assign 'looping:=true'/>
    <while condition= 'looping'>
      <sequence>
        <invoke select/>
        <switch>
          <case condition= 'sc'>
            <invoke configure/>
          </case>
        </switch>
        <switch>
          <case condition= 'sp | (sc & cp)'>
            <invoke put/>
          </case>
        </switch>
        <assign 'looping:=ss | (sc & cs) | ((sp | (sc & cp)) & ps)'/>
      </sequence>
    </while>
    <switch>
      <case condition= 'sM | (sc & cM) | ((sp | (sc & cp)) & pM)'>
        <invoke merge/>
      </case>
    </switch>
  </sequence>
</process>
```

Figure 14
Generated abstract BPEL4WS code for region R3

that need to be addressed in using such an approach. Apart from the impossibility of determining contradictions and termination (in the general case), the bidirectional nature of the specification does not address transformations that are not a (mathematical) function in a direction, that is, those which have more than one possible solution. In theory, one could make a random choice as to which solution to apply, but in practice, often only one or a subset of the solutions is desirable.

OCL also has a limitation in expressing complex transformations. In this context, OCL expressions can become large, cumbersome, and very difficult to write, debug, and understand. In particular, mappings that are not one-to-one and that are resolved only with complex or intricate computations can be difficult to express. Some improvements can be made by changes to syntax, but most, at least the deep-rooted issues, require a different approach for expression. This limitation points out the problems of many declarative approaches in this context, which has led to a movement toward hybrid languages, which mix declarative and imperative approaches.[29]

SUMMARY AND OUTLOOK

We have presented declarative transformations that enable the generation of executable workflow code specified in BPEL4WS from the graphical representation of control-flow diagrams. The transformations themselves are based on control-flow analysis and normalization techniques, which originate from the field of compiler theory. In this paper, we have investigated their declarative representation using OCL.

The results we have obtained are encouraging. Complex transformations can be declaratively described when means to reduce the complexity of declarative statements are provided. These involve a readable concrete syntax and the ability to introduce abbreviations for frequently occurring subexpressions and to reuse and embed specialized transformations that target specific transformation subproblems.

We are following two main avenues of future research. We are working on extending our transformation methods to more complex control-flow diagrams that combine concurrent and cyclic flows in a more general way. Such an extension is not straightforward because it requires that the semantics of general UML 2.0 activity diagrams be strictly formalized. Although initial formalizations exist, they do not yet capture the whole expressivity of the representation; and existing tools, which could detect inconsistent models, for example, those containing flows with deadlocks or "livelocks," are not very scalable.

At the same time, we are working on a solution for model reconciliation that will allow us to keep business-level and IT-level models synchronized while they are undergoing change and to execute automatic transformations in both directions. Such transformations have a high practical relevance in areas such as business activity monitoring. Current monitoring tools provide users with a monitoring of the running IT-level workflows and can only propagate the monitoring results back to the business-level models if a straightforward one-to-one mapping between both models exist. How monitoring results can be propagated back for models that have undergone a major transformational change as discussed in this paper is currently an unsolved problem.

ACKNOWLEDGMENTS

We thank Achim Brucker, Jochen Küster, John Novatnack, and James Rumbaugh for their important feedback on earlier versions of the paper.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Object Management Group, Inc., or Sun Microsystems, Inc.

CITED REFERENCES AND NOTE

1. *Unified Modeling Language Superstructure 2.0 Specification,* OMG document ptc/03-08-02 (2003), http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.pdf.
2. S. Thatte et al., "Business Process Execution Language for Web Services Version 1.1," (2003), http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.
3. J. Warmer and A. Kleppe, *The Object Constraint Language,* Second Edition, Addison-Wesley, Reading, MA (2003).
4. M. Hecht and J. Ullman, "Flow graph reducibility," *SIAM Journal of Computing* **1,** No. 2, 188–202 (1972).
5. P. Krill, "IBM Adds to Business Process Modeling Tool," Infoworld.com (August 10, 2004).
6. E. Deborin, J. Basrai, T. Benedetti, R. Halchin, T. Mahfouz, N. Perera, B. S. Shamshabad, R. Spory, and R. Turakhia, *Continuous Business Process Management with HOLOSOFX BPM Suite and IBM MQSeries Workflow,* IBM Redbooks (2002), http://www.redbooks.ibm.com/abstracts/sg246590.html?Open.
7. A. W. Scheer, F. Abolhassan, W. Jost, and M. Kirchner, *Business Process Excellence—ARIS in Practice,* Springer-Verlag (2002).
8. D. K. Taft, "Microsoft Zeros in on Model-based Programming," *eWeek* (July 24, 2004).
9. W. van der Aalst, J. Desel, and E. Kindler, "On the Semantics of EPCs: A Vicious Circle," *Proceedings of the Workshop on EPK,* GI-Arbeitskreis Gesch (2003), pp. 7–18.
10. A. van Dorp et al., *WS-I Web Enterprise Customer Commerce System—Business Scenario,* Web Services Interoperability Organization, WS-I Technical Document (2003).
11. C. Bock, "UML 2 Activity and Action Models," *Journal of Object Technology* **2,** No. 4, 43–53 (2003).
12. H. Stoerrle, "Semantics and Verification of Data flow in UML 2.0 Activity Diagrams," *Proceedings of the IEEE Workshop on Visual Languages and Formal Methods* (to be published).
13. R. Eshuis and R. Wieringa, "Verification Support for Workflow Design with UML Activity Graphs," *Proceedings of the 24th Conference on Software Engineering,* IEEE Press, NY (2002), pp. 166–176.
14. E. Boerger, A. Cavarra, and E. Riccobene, "An ASM Semantics for UML Activity Diagrams," in *Algebraic Methodology and Software Technology,* T. Rust, editor, Volume LNCS 1816, Springer-Verlag (2000), pp. 292–308.
15. Y. Dong and Z. ShenSheng, "Using Pi-calculus to Formalize UML Activity Diagrams," *Proceedings of the 10th International Conference and Workshop on the Engineering of Computer-based Systems,* IEEE Press, NY (2003), pp. 47–54.
16. X. Fu, T. Bultan, and J. Su, "Model Checking XML Manipulating Software," *Proceedings of 2004 International ACM/SIGSOFT International Symposium on Software Testing and Analysis,* ACM Press, NY (2004), pp. 252–262.
17. A. Martens, C. Stahl, D. Weinberg, D. Fahland, and T. Heidinger, *Business Process Execution Language for Web Services—Semantics, Analysis, and Visualization,* Technical Report 169 (in German), Humboldt University, Berlin (2004).
18. T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie, *Zing: A Model Checker for Concurrent Software,* Technical Report MSR-TR-2004-10, Microsoft Research (2004).
19. F. Allen, "Control Flow Analysis," *ACM SIGPLAN Notices* **5,** No. 7, 1–19 (1970).
20. A. Aho, R. Sethi, and J. Ullman, *Compilers—Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA (1986).
21. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM* **9,** No. 5, 366–371 (1966).
22. W. Peterson, T. Kasami, and N. Tokura, "On the Capabilities of while, repeat, and exit Statements," *Communications of the ACM* **16,** No. 8, 503–512 (1973).
23. R. Hauser and J. Koehler, "Compiling Process Graphs into Executable Code," *Proceedings of the Third Interna-*

tional Conference on Generative Programming and Component Engineering, LNCS Volume 3286, pp. 317–336, Springer-Verlag (2004).

24. Z. Ammarguellat, "A Control-Flow Normalization Algorithm and Its Complexity," *Software Engineering* **13,** No. 8, 237–251 (1992).
25. P. Nandi, S. Kumaran, J. Chung, T. Heath, R. Das, and K. Bhaskaran, "ADoc-Oriented Programming," *Proceedings of the 2003 International Symposium on Applications and the Internet* (2003).
26. An "ADoc" is an adaptive document. ADoc-oriented programming was proposed as a programming model based on object-oriented principles for assembling solutions with emphasis on collaboration and integration. The ADoc is a higher-level modeling artifact with adaptive behavior similar to a state machine.
27. J. Koehler and R. Hauser, "Untangling Unstructured Cyclic Flows—a Solution Based on Continuations," *Proceedings of the International Conference on Cooperative Information Systems,* LNCS Volume 3290, pp. 121–138, Springer-Verlag (2004).
28. *MOF 2.0 Query/Views/Transformations RFP,* OMG document ad/02-04-10 (2002), http://doc.omg.org/ad/2002-4-10.
29. T. Gardner, C. Griffin, J. Koehler, and R. Hauser, *A Review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the Final Standard,* OMG document ad/03-08-02 (2003), http://www.omg.org/docs/ad/03-08-02.pdf.
30. *MOF 2.0 Query/Views/Transformations RFP* (revised submission), OMG document ad/04-04-01 (2004), http://doc.omg.org/ad/2004-04-01.
31. G. Sunye, D. Pollet, Y. Le Traon, and J.-M. Jezequel, "Refactoring UML Models," *Proceedings of UML 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference,* LNCS Volume 2185, pp. 134–148, Springer-Verlag (2001).
32. D. Pollet, D. Vojtisek, and J.-M. Jezequel, "OCL as a Core UML Transformation Language," *Workshop on Integration and Transformation of UML models* (WITUML 2002), Position paper (2002).
33. A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture—Practice and Promise,* Addison-Wesley, Reading, MA (2003).
34. *XMOF Queries, Views and Transformations on Models Using MOF, OCL and Patterns,* OMG document ad/2003-08-07 (2003), http://www.omg.org/cgi-bin/apps/doc?ad/03-08-07.pdf.

Accepted for publication August 20, 2004
Internet publication January 7, 2005.

Jana Koehler
IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (koe@zurich.ibm.com). Dr. Koehler is a research staff member and project leader in the Computer Science department at the IBM Zurich Research Laboratory. She received an M.S. degree in computer science and the science of science from Humboldt University Berlin, Germany, in 1988, a Ph.D. degree in computer science from the University of Saarbrücken, Germany in 1994, and her habilitation in computer science from the University of Freiburg, Germany, in 1999. She joined IBM at the Zurich Research Laboratory in 2001 after being an assistant professor from 1996–1999 and a project leader in technology management for Schindler AG, Switzerland, from 1999–2001. She is the winner of several scientific and best paper awards. Dr. Koehler is a member of the German Informatics Society and the American Association of Artificial Intelligence.

Rainer Hauser
IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (rfh@zurich.ibm.com). Dr. Hauser is a research staff member in the Computer Science department at the IBM Zurich Research Laboratory. He received a diploma in mathematics in 1977 and a Ph.D. degree in computer science in 1984 from the Swiss Federal Institute of Technology (ETH). He joined IBM at the Zurich Research Laboratory in 1980, initially as a Ph.D. student working on image processing, and after completion of the Ph.D. degree, joined a team working on communication systems.

Shane Sendall
IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (sse@zurich.ibm.com). Dr. Sendall is a post-doctoral researcher in the Computer Science department at the IBM Zurich Research Labs (ZRL). He received B.S. and B.S. (Honors) degrees in computer science from University of Queensland in 1996 and 1997 and a Ph.D. degree in software engineering from the Swiss Federal Institute of Technology in 2002. Dr. Sendall was subsequently a member of the Software Modeling and Verification Laboratory at the University of Geneva until 2004. Thereafter, he joined IBM at ZRL, where he is working on model transformation in the context of software development tools and practice. He is a member of the Institute of Electrical and Electronics Engineers and the Association for Computing Machinery.

Michael Wahler
IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (wah@zurich.ibm.com). Mr. Wahler is a doctoral student in the e-Business Solutions department at the IBM Zurich Research Laboratory. He received a diploma in computer science from Technische Universität München in 2003. He subsequently joined the Business Process Integration and Automation (BPIA) project, where he is working on formalizations of model transformation approaches. ■