A Model-Driven Transformation Method

Jana Koehler Rainer Hauser
IBM Zurich Research Laboratory
CH-8803 Rueschlikon
Switzerland
email: {koe | rfh}@zurich.ibm.com

Shubir Kapoor Fred Y. Wu Santhosh Kumaran
IBM T.J. Watson Research Center
PO BOX 218 RTE 134
Yorktown Heights, NY 10598, USA
email:{ shubirk | fywu | sbk}@us.ibm.com

Abstract

Model-driven architectures (MDA) separate the business or application logic from the underlying platform technology and represent this logic with precise semantic models. These models are supposed to span the entire life cycle of a software system and ease the software production and maintenance tasks. Consequently, tools will be needed that support these tasks.

In this paper, we present a method that implements model-driven transformations between particular platform-independent (business view) and platform-specific (IT architectural) models. On the business level, we focus on business view models expressed in ADF or UML2, whereas on the IT architecture side we focus on service-oriented architectures with Web service interfaces and processes specified in business process protocol languages such as BPEL4WS.

1 Introduction

Model-driven architectures (MDA) have been proposed by the OMG to reinforce the use of an enterprise architecture strategy and to enhance the efficiency of software development. The key to MDA lies in three factors:

- the adoption of precise and complete semantic models to specify the structure of a system and its behavior,
- the usage of data representation interchange standards,
- the separation of business or application logic from the underlying platform technology.

Each of these key factors poses a challenge of its own, but the model representation seems to be the most critical among them. Models can be specified from different views, from the business analyst's or the IT architect's view, and they can be represented at different levels of abstraction. MDA distinguishes between platform-independent (PIM) and platform-specific (PSM) models, but we can expect that many different PIM and PSM models will be used to describe the different aspects of a system. Representing a model of a system (be it a PIM or a PSM) is only one side of the coin. In order to be useful, a model must be further analyzable by algorithmic methods. For example, the fundamental properties of the behavior of a system should be verifiable in a model, and different models of the same system should be linked in a way that relevant changes in one model can be propagated to the other models to keep the model set consistent.

In this paper, we develop a method to transform particular platform-independent into particular platform-specific models. On the PIM side, we use *business process models* that reflect a business analyst's or consultant's view of a business process. Typical representatives of these models are the ARIS tool set by IDS Scheer[15], the ADF approach by Holosofx/IBM [4] or UML2 activity models [17]. On the PSM side, we distinguish between *IT architectural models* and *IT deployment models*. The IT architectural models that we generate are service-oriented and focus on a specific programming (not hardware) platform using Web services [2] and workflows [8]. Therefore, we consider them to be PSMs. The runtime-platform specific information is captured in the IT deployment model, which further refines the architectural model, and, for example, provides the port and binding information for Web services or the physical deployment of a workflow on a particular workflow engine. We do not consider IT deployment models in this paper.

It is generally acknowledged that an IT architecture should be aligned with the business processes and it should be the business process model that determines the IT architecture and not the other way round. In this sense, one can regard the PIM as the specification and the PSM as a solution satisfying the specification. Since model-driven architectures aim at spanning *"the life cycle of a system from modeling and design to component construction, assembly, integration, deployment, management and evolution"* [9]

we can expect that

1. several different models will exist that describe a system from different points of view using different levels of abstraction and
2. the models will evolve following the life cycle of a system.

Therefore, it seems necessary that algorithmic techniques be developed and tools be provided that support the evolution of models and the consistency of different modeling views of the same system. In this paper, we present initial results in this direction and discuss some of the methods that we developed for a prototype tool—called the transformation engine (TE)—that addresses these tasks.

The paper is organized as follows: In Section 2, we discuss the role of model-driven transformations and present the architecture of our transformation engine. Section 3 reviews different business process modeling approaches and extracts structured information flows as their underlying common modeling principle. Section 4 summarizes the IT architectural models that we focus on and presents methods to synthesize selected IT architectural solution components. In Section 5, we discuss the correctness of model-driven transformations. In Section 6, we review related work and conclude with a summary and an outlook on the challenge of dynamic model reconciliation in Section 7.

2 Role of Model-Driven Transformations

We have currently developed *top-down* and *bottom-up* model-driven transformations between particular PIM and PSM. The *top-down transformation* starts from the business view model of a system or process as the PIM and derives a platform-specific IT architecture model, the PSM. This ensures that the business model determines the relevant elements of the IT architecture. It transforms the blueprint of a system as it is envisioned during a business analysis and re-engineering process into an IT architecture that will implement that blueprint.

The *bottom-up transformation* takes a platform-specific IT architecture model as input and abstracts it into a business view model. It allows the propagation of structural changes in the IT architectural model back to the business view model and thereby supports communication from IT people to business people.

Both transformations prolong the life cycle of the business view model. It is no longer only actively used in the design and analysis phase, but plays an active role in the entire system life cycle by constraining the IT architecture. The transformation engine has been designed to transform different business view models automatically into IT architectural models and vice versa. At both sides, a variety of approaches with particular strengths and weaknesses compete with each other. Therefore, we introduced model-independent layers into the TE architecture. This allows us to handle a variety of PIMs and PSMs, see Figure 1.

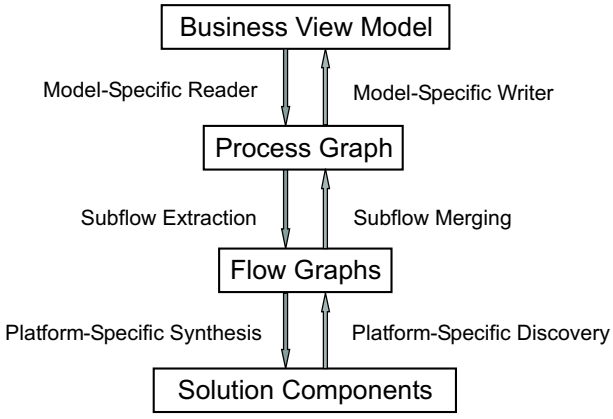

Figure 1. Architecture of the Transformation Engine

On the business view side, the TE can handle modeling approaches that structure a business process into atomic process steps and subprocesses. Atomic process steps are not further refined, whereas subprocesses can be nested within each other and lead to a hierarchical process model. The process elements (subprocesses, process steps) are linked by typed information entities that represent the flow of control and physical data objects or event messages between the process elements. ADF [4], ARIS [15], UML2 activity models [17], and Business Operational Specifications (BOpS), a yet unpublished modeling approach by IBM Research, are typical representatives of these modeling approaches. They all support hierarchical process models that describe processes as *structured information flows* across *distinguishable* process elements.[1]

The transformation engine contains specific readers and writers for each modeling approach. When performing a top-down transformation, the reader reads in a specific business view model and maps it to a so-called *process graph*, which we have developed as a generalization of the information-flow models characterized above. The process graph is then further analyzed to extract the *information type-specific subflows* and then decomposed into *flow graphs* representing these subflows. The actual transformation therefore takes place between the process graph on one side and and the flow graphs on the other side. From the flow graphs, an IT architecture-specific synthesis method generates solution components that are required in the de-

[1] The models we consider in this paper only comprise a single process-oriented view. Work on transformation methods for multi-view models (a typical example is the OMG Enterprise Collaboration Architecture) is in progress.

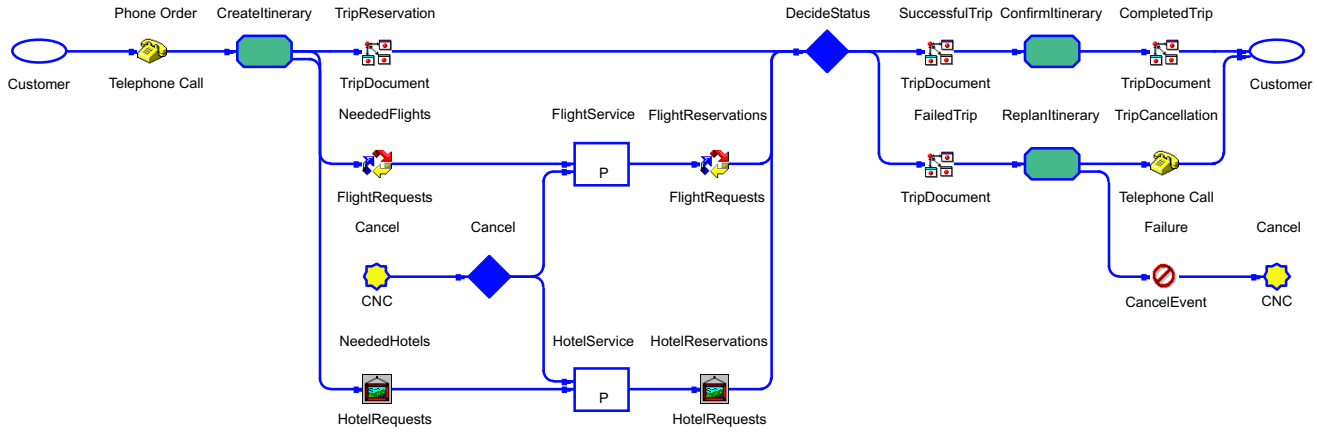

Figure 2. ADF representation of the trip-handling process.

sired IT architectural model. In this paper, we discuss the generation of *Adaptive Documents* (ADocs) [10], which describe persistent business documents whose life cycle is controlled by an associated state machine, and business protocols between Web services (specified as BPEL4WS [3] processes). We do not discuss the generation of the Web service interfaces for the solution components, nor do we consider additional solution components such as screenflows, connectors, business objects, or others, and the "wiring" of these components with each other.

When performing bottom-up transformations, the TE is provided with an IT architectural model comprising a set of solution components from which it synthesizes the flow graphs that are merged into a single process graph. This graph can then be mapped to a specific business view model using a model-specific writer. In this paper, we will limit our discussion to the top-down transformations, which we developed.

3 Business Models based on Structured Information Flows

Let us introduce our generalization of business view models to process graphs with the help of an example, namely the trip-handling process of a travel agency. The agency receives trip requests from customers over the phone. The requests are recorded in trip documents, from which several flight and hotel requests are derived by the agent. The flight and hotel requests are handled by specialized services that receive these requests from the travel agency. The services report their success or failure in booking the requests back to the agency. If all requests were successfully handled, the agency puts together the trip documents for the customer and confirms the travel itinerary. If one of the services fails to make a requested reservation, the agency immediately informs the other service to put all reservation requests on hold and contacts the customer to revise the trip itinerary.

Figure 2 shows an ADF model of this process. The customer is an external entity (depicted with a white oval) initiating the trip-handling process. Three main elements are used to capture the structure of the process: (1) Grey octogons depict the elementary process steps, which are named as *CreateItinerary*, *ConfirmItinerary*, and *ReplanItinerary*. (2) White boxes containing a "P" letter depict subprocesses and refer to the *FlightService* and *HotelService*. (3) Black diamonds show binary decision steps in the process.

Typed information entities flow between the various process steps. In the example model, we distinguish between five different types of information entities named *TelephoneCall*, *TripDocument*, *FlightRequests*, *HotelRequests*, and *CancelEvent*. Each of them is depicted with a separate so-called Phi symbol. Below each Phi symbol, we find the type and above the symbol, we find the Phi name (the particular instance of this type).

A star-shaped goto symbol allows to model cyclic processes. In our example, the Phi *Failure* of type *CancelEvent* is sent from the *ReplanItinerary* step back to the *FlightService* and *HotelService* subprocesses via the CNC goto symbol. Which of the services has to be notified is decided in the decision step receiving the Phi.

The UML2 activity diagram in Figure 3 shows another model that we created for our example process. It models the customer and the trip-handling process as two activities, which are linked via flows of the order event (OE), trip request (TR), and booking failure (BF) token. The trip-handling activity is further structured into actions that we already distinguished as process steps in the ADF model. The *Flight-* and *HotelService* activities are contained within an interruptible region. They are linked to the other actions via the flow of flight request (FR) and hotel request (HR) tokens and may be interrupted by a booking failure token.

A merge node combining TR, FR and HR tokens and generating a new TR token and a decision node, which decides about the failure or success of a trip reservation, are also contained in the model.

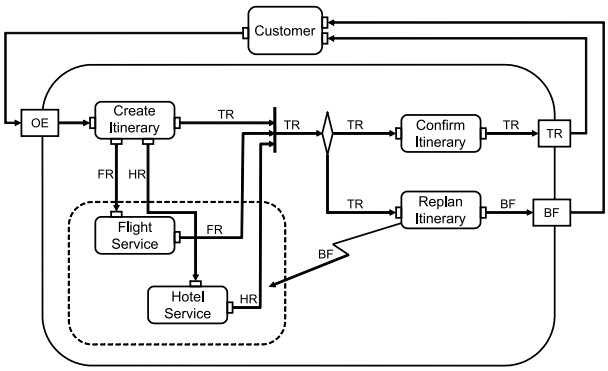

Figure 3. UML2 representation of the trip-handling process.

The two modeling approaches have several features in common. First, they structure a process into different process elements:

- *subprocesses*, which can optionally be further refined and which are either *internal* or *external* given the current view of the process,
- atomic *process steps* that actively produce, or receive and modify one or more information entities, and
- *control steps* that inspect and merge information entities and represent the decision logic of a process depending on the information content.

Second, both models capture the flow of information entities, which can be object documents of a particular type, events, or any other messages. Arbitrary types of information entities can be defined by a modeler and then used to annotate links between the process elements. Table 4 summarizes the specific elements from each modeling approach and how they fit into the categorization as subprocesses, process steps, control steps, and information entities.

Neither of the two graphical representations is really unambiguous in the sense that a human (or a program) could derive insights into the dynamic behavior of the process merely by looking at the graphical (structural) models. Without the informal process description given above, different readers would probably come up with quite different interpretations of the models regarding details of the dynamic behavior of the process. Consequently, both approaches provide many ways of annotating process models with additional information. For example, UML2 allows a

	ADF	UML2
subprocess	process external task external entity	activity
process step	task	action
control step	decision choice	decision, merge fork, join
information entity	Phi	token
flow	connector stop goto	link initial marker final marker

Figure 4. Categorization of modeling elements in ADF and UML2.

modeler to describe the pre- and postconditions of actions to further illustrate possible process behaviors, but leaves the choice of language to formulate these conditions to the human modeler.

These (and other) approaches to business process modeling have been developed with the goal to support a group of designers and analysts, who need to achieve a common understanding of the process behavior. Many details of the process behavior are thus either not visible in the model representation or contained in natural language descriptions that are inaccessible to an automatic analysis tool. Even if dynamic behavior is sketched in the model, for example the interruptible region that we identified in the UML2 model, our example models are not detailed enough to enable a tool or a user to derive an unambiguous interpretation of the process behavior. The only unambiguous information that the models provide is the specification of the *static* flow of typed information entities between structural process elements. In this paper, we investigate what a transformation method can accomplish when limited to consider only such static information flows, because these can be automatically extracted from a model without facing a high risk of misinterpreting the business view model. We developed an intermediate representation called a *process graph* that allows us to

- map the static information flows represented differently in different modeling approaches to the same unambigous representation
- assign a formal, state-based operational semantics to the process graph representation that is exploited by the transformation methods.

3.1 Process Graphs

A process graph $G = (V, E, T)$ is a formal model to represent structured information flows.

Definition 1 *Let $G = (V, E, T)$ be a directed graph with vertices V, and edges E, and let T be a not further specified type system.*

- *$V = V_S \cup V_C \cup V_E \cup V_I$ is the set of vertices with V_S being the process steps, V_C the control steps, V_E the abstract external processes, and V_I the abstract internal processes.*
- *$E = (E_O \cup E_E) \subseteq V \times V \times T$, with $T = T_O \cup T_E$, are the typed and directed edges of the graph with E_O being the object edges (carrying objects of type T_O) and E_E the event edges (carrying events of Type T_E).*

The set of type names denotes types of distinguished information entities that are input and output of the vertices in the graph. We only consider two disjoint sets of types to denote events and objects as the major two groups of information entities. Both type sets can be further refined by arbitrary model-specific subtypes.

Definition 2 *A vertex $v \in V$ has a signature $\Sigma = \Sigma_I \cup \Sigma_O$ of type names t_i from a given type system T. We distinguish in Σ the input signature $\Sigma_I = \{t_m, \ldots, t_n\}$ and the output signature $\Sigma_O = \{t_k, \ldots, t_l\}$.*

Definition 3 *An object constant is a name to denote an object that is described with a (possibly empty) set of properties specified in some language L. An assignment of values to these properties is the state of the object.*
An event constant is a name to denote a not further described, stateless object. Each object or event constant has a specific object or event type.

Process steps as well as abstract internal and external subprocesses can change the state of one or more objects, whereas control steps cannot. Internal and external processes are only distinguished from process steps to represent the process boundaries that are often identified in business view models. Otherwise, their semantics is the same, i.e., they can change the state of objects.

Definition 4 *A typed directed edge $e = (v_i, v_j, t_k)$ from vertex v_i to vertex v_j can be added to the graph if $t_k \in \Sigma_O(v_i) \cap \Sigma_I(v_j)$, i.e., the type must exist in the output signature of the vertex where the edge starts and in the input signature of the vertex where it ends. Note that $i \neq j$ must not necessarily hold.*

Edges establish a typed link from one vertex to another to show the transfer of information entities between process elements. The type of the information entity does not change during the transfer. The edges define paths across the vertices of the graph and induce a partial ordering of the vertex elements.

Definition 5 *A vertex v_i is ordered before a vertex v_j (written $v_i \prec v_j$, $i \neq j$) if there is a path from v_i to v_j, but no path from v_j to v_i.*

There are two cases where vertices can remain unordered with each other: either there is no path between them at all or they lie on a cycle in the graph. Neither of these cases is a problem for our approach.

3.2 Building and Analyzing Process Graphs

A process graph only depicts the structure of the process and the principal information flow between process elements, i.e., it is a very abstract process model that does not allow the study of the dynamic behavior of the process. Figure 5 shows the process graph that we generate from the UML2 and ADF example models. The symbols TR, OE, HR, FR, BF are the token names from the UML2 model that we reused to name the information entities in the process graph.

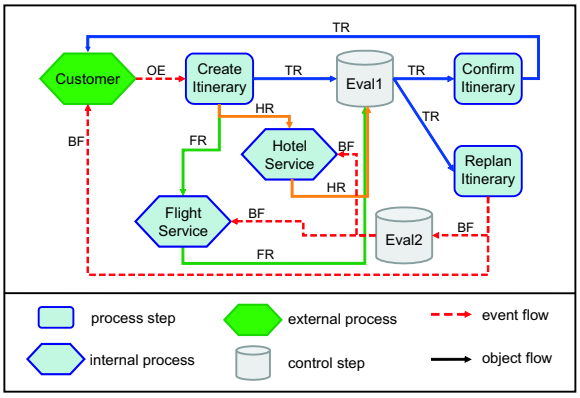

Figure 5. Generalization of business view models in the process graph.

For each modeling approach, a specific reader analyzes the model description and extracts model elements that can be mapped to the process graph elements following the basic mapping principles shown in Figure 4. We use a repository symbol to depict control steps in order to emphasize that control steps inspect the state of objects, but do not change them (in contrast to the other process elements). The interruptible region in the UML2 activity model and the lightning arrow from the *ReplanItinerary* process step into this region are mapped to a new subgraph containing an event link from this process step to a new control step, from which event links lead to all process steps or subprocesses in the region. If a model establishes a link between two process steps without assigning to it any information entity, which is usually known as a control link, then we generate

a new event constant and map the control link to an event link in the process graph. This guarantees that the ordering between two activities that are linked with the control link is preserved in the process graph via the event link.

Note that a process graph is flat, i.e., it has no hierarchical structure. If a business view model contains a hierarchical process model where a subprocess at a higher level is refined with a separate process model at a lower level, then the subprocess vertex in the process graph can be replaced by the process graph obtained from the lower-level process model if desired. It is up to the modeler to decide whether subprocesses should be kept as abstract process steps or refined into their subprocesses. The replacement only succeeds if the signature of the subprocess vertex is identical to the signature of the subprocess graph, which is the declaration of incoming and outgoing information entities for the subprocess. The model-specific readers follow very closely the informal semantics as it is described in the modeling guidelines of the various approaches, but obviously we cannot establish a formal equivalence guarantee of the process graph with the original model.

After the process graph has been constructed, it becomes input to the transformation methods, which extract various subgraphs from it:

- The subgraphs specifying the flow of one object type between process steps and internal subprocesses shows the process-internal workflow of this particular object type.
- The subgraphs containing an external subprocess vertex and all process-step vertices that have direct edges to this external subprocess vertex show the interface of a process to its external partners. The signature of the external subprocess vertex indicates the message types that are exchanged between the partners. Similarly, the interface of a process to its internal partners can be determined by extracting the corresponding subgraphs for the internal subprocess vertices.
- Flows of one object type that spawn more than one internal process step indicate stateful business objects.

Figure 6 shows the individual subflows for the three object types (trip request, hotel request, flight request) that we discovered in the trip handling example.

Figure 7 shows the directed information flows between process steps (white rectangle or diamond) and external or internal subprocesses (grey hexagon).

Even with very little information only sketching the static information flow, we can still derive useful IT architectural components. In the following, we will demonstrate how the subflow graphs will be mapped to selected solution components of an IT architecture by platform-specific synthesis methods.

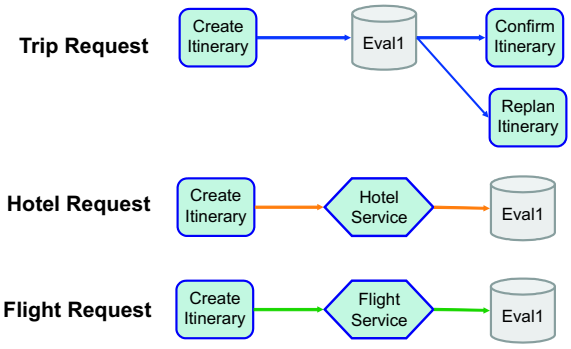

Figure 6. Three detected object types and their processing flows.

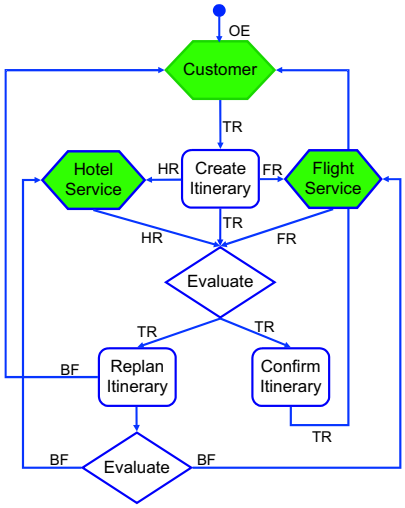

Figure 7. Information flow dependencies between the steps of the trip-handling process and its partners.

4 IT Architectural Models and the Automatic Synthesis of Solution Components

Our target IT architecture is based on Web services—self-contained, modular units of application logic which provide business functionality to other applications via an Internet connection. In a nutshell, a *Web service* is specified by defining messages that provide an abstract definition of the data being transmitted and operations that a Web service provides to transmit the messages. Operations are grouped into port types, which describe abstract end points of a Web service such as a logical address under which an operation can be invoked. Concrete protocol bindings and physical address port specifications complete a Web service specification.

Web services support the interaction of business partners and their processes by providing a stateless model of "atomic" synchronous or asynchronous message exchanges. These "atomic" message exchanges can be composed into longer business interactions by providing message exchange protocols that show the mutually visible message exchange behavior of each of the partners involved. An example of such a protocol specification language is BPEL4WS [3]. In the following, we show how the basic elements of a BPEL4WS protocol can be derived from the process graph.

4.1 Synthesis of BPEL4WS Protocols

The starting point for the synthesis of a BPEL4WS protocol is the information contained in Figure 7. It shows the trip-handling process with its individual process steps (all white-colored vertices) and its three communicating partners: (1) the customer who is an external partner to the travel agent (top grey hexagon), (2) the flight service which is an internal partner (right grey hexagon), i.e., a subprocess that we have not yet further specified, and finally (3) the hotel service which is also an internal partner dealing with the hotel request processing (left grey hexagon). This information about partners is mapped to the partner declaration in BPEL4WS. To make the example more interesting, we assume that the interaction with all partners proceeds via a Web service interface.

```
<process name ="TripHandling">
  <partners>
    <partner name ="Customer"
           myRole ="TripHandlingAgent"
  serviceLinkType ="CustomerServiceLink"
      partnerRole ="CustomerAgent"/>

    <partner name ="FlightService"
           myRole ="TripHandlingAgent"
  serviceLinkType ="FlightServiceLink"
      partnerRole ="FlightServiceAgent"/>

    <partner name ="HotelService"
           myRole ="tripHandlingAgent"
  serviceLinkType ="HotelServiceLink"
      partnerRole ="HotelServiceAgent"/>
  </partners>
   ...
</process>
```

The name of the process is derived from the name of the business view model. The names of the partners are mechanically derived from the external and internal subprocess names. The partner role names are composed of the partner names with the suffix `Agent` added, whereas the value of the attribute `myRole` is again the model name plus the `Agent` suffix. The names for the service link types are also mechanically derived from the partner names.

For each information entity that we discovered in the business view model and that occurs in the signature of at least one process step in the trip-handling process, a message name and corresponding container name is defined.

```
<containers>
  <container name ="OrderEvent"
      messageType ="OrderEventType"/>
  <container name ="TripRequest"
      messageType ="TripRequestType"/>
  <container name ="FlightRequests"
      messageType ="FlightRequestType"/>
  <container name ="HotelRequests"
      messageType ="HotelRequestType"/>
  <container name ="BookingFailure"
      messageType ="BookingFailureType"/>
</containers>
```

The most interesting task, but also the greatest challenge, is to derive the control flow of the message exchange between the process and its partners. The directed edges between the vertices in the graph as shown in Figure 7 provide the required information. Directed edges between two internal process steps provide ordering information, i.e., in the example *CreateItinerary* comes before *Evaluate*, *Evaluate* comes before *ReplanItinerary* and *ConfirmItinerary*, but the latter two are unordered with each other. All steps proceed before the second *Evaluate*. This information yields the control structures for the BPEL4WS specification. Steps that are ordered with each other are put within a `<sequence>` activity, while unordered steps are put within a `<flow>` activity. After each control step, a `<switch>` activity is introduced.

```
<sequence>
  createItinerary
  Evaluate
  <switch>
    ConfirmItinerary
    <otherwise>
       <sequence>
         ReplanItinerary
         Evaluate
         <switch>
           HotelService
           <otherwise>
             FlightService
           </otherwise>
         </switch>
       </sequence>
    </otherwise>
  </switch>
</sequence>
```

In a second step, the mnemonic step names have to be replaced with the message interactions a step has with a partner. Each directed link that links a step vertex with a partner vertex is mapped to a basic BPEL4WS activity. A directed edge from a partner vertex to a step vertex is interpreted as a message flow from a partner to the process. We map it therefore to a `<receive>` activity. A directed edge

into the other direction, i.e., from a step vertex to a partner vertex is mapped to an <invoke> activity.

The process starts with the *CreateItinerary* step receiving an order event message from the customer. By default, we always instantiate a new process instance. After the message has been received, the trip request object is created, and hotel and flight request messages can be sent in any order to the two partner services. Consequently, both <invoke> activities are embedded into a <flow> activity. Port type names are simple default strings and operation names are derived from the (abbreviated) names of the interacting process elements. For example, *Customer to Create Itinerary* is abbreviated with CToCI, *Create Itinerary to Flight Service* is abbreviated with CIToFS, the control steps are abbreviated with EVAL.

```
<sequence>
  <receive partner="Customer"
         portType ="pt1"
        operation ="CToCI"
   createInstance ="yes"
        container ="OrderEvent">
  </receive>

  <flow>
    <invoke partner ="HotelService"
           portType ="pt2"
          operation ="CIToHS"
     inputContainer ="HotelRequests">
    </invoke>

    <invoke partner ="FlightService"
           portType ="pt3"
          operation ="CIToFS"
     inputContainer ="FlightRequests">
    </invoke>
  </flow>
```

After the partner services have been invoked, the process waits for the services to send the results of their booking operations, which again can arrive in any order.

```
 <flow>
    <receive partner ="HotelService"
      portType ="pt4"
      operation ="HSToEVAL1"
      container ="HotelRequests">
   </receive>

  <receive partner ="FlightService"
          portType ="pt5"
         operation ="FSToEVAL1"
         container ="FlightRequests">
  </receive>
</flow>
```

After the answers have been received, the process has to branch depending on whether the services were able to book the requests or not. This introduces a first <switch> construct in the process. No condition for the switching can be derived from the process model and therefore a default placeholder is inserted. In the first branch (*ConfirmItinerary to Customer*), the process needs to inform the customer about the successful completion of his reservation and sends the completed trip request documents. In the second branch (*ReplanItinerary to Customer, Evaluate to HotelService or FlightService*), it needs to inform the customer about the booking failure and then decide which of the services has to be informed about the failure of the other partner, i.e., another <switch> construct is nested that decides which of the services is informed.

```
<switch>
  <case condition ="condition1">
    <invoke partner ="Customer"
           portType ="pt6"
          operation ="ConIToC"
     inputContainer ="TripRequest">
    </invoke>
  </case>
  <otherwise>
    <sequence>
      <invoke partner ="Customer"
             portType ="pt7"
            operation ="RIToC"
       inputContainer ="BookingFailure">
      </invoke>
      <switch>
        <case condition="condition2">
          <invoke partner ="HotelService"
                 portType ="pt8"
                operation ="EVAL2ToHS"
           inputContainer ="BookingFailure">
          </invoke>
        </case>
        <otherwise>
          <invoke partner ="FlightService"
                 portType ="pt9"
                operation ="EVAL2ToFS"
           inputContainer ="BookingFailure">
          </invoke>
        </otherwise>
      </switch>
    <sequence>
  </otherwise>
</switch>
</sequence>
```

By default, we assume that all message exchanges are asynchronous. The BPEL4WS specification becomes rather complicated due to the two control steps contained within the process. Obviously, these could be merged into a single <switch> construct with three different cases by applying BPEL4WS optimization techniques similar to the ones presented in [14]. The transformation engine could then propagate such simplifications back to the business view model where it could be discussed with the process designers.

An alternative transformation method (not shown in this paper) generates an unordered set of <receive> and <invoke> activities within a single flow in the same way as above. Then it adds <links> from an activity A to an activity B whenever there is a path from the process step

interacting with A to the process step interacting with B in the process graph.

4.2 Synthesis of State Machines

Another programming model that we want to support with our synthesis method are stateful business objects. So-called *adaptive documents* (ADocs) [10], are persistent data objects that carry their own processing instructions in the form of an associated (possibly nondeterministic) finite-state machine. When in a particular state, an ADoc can invoke an activity and, depending on the result of the activity execution, it will transit to one of the possible successor states. The task of the synthesis process is to generate state machine specifications for each of the discovered business objects.

In our example, we have discovered three different business objects, namely the trip request, the flight request, and the hotel request. Each of them is processed by different activities reflected in the subprocesses and process steps of the information flows that we extracted from the process graph, see Figure 6. Now, we use this information to determine the state machine of the objects.

We consider the flow graph of each object as a means of defining the words of a formal language L_O. The names of the vertices are the symbols of the language, and the edges define the allowed concatenation of the symbols to obtain a word in the language L_O. To simplify our notation we abbreviate the activity names with single letters: *CreateItinerary* (c), *Evaluate* (e), *HotelService* (h), *FlightService* (f), *ReplanItinerary* (r), and *ConfirmItinerary* (o). The flow graph for the trip request object defines the words $ce(o \mid r)$. For the hotel request object we obtain che, and for the flight request object we have cfe. We now construct three finite state machines that accept these words.

The construction process defines a transition for each graph vertex and annotates it with the name of the vertex to indicate the activity that has to be invoked. For each transition, a start and an end state are defined. When a process step A is directly followed by a process step B in the graph, the end state of A is merged with the start state of B. This way, the state machines are built until the end state of the last process step is marked as a final state of the machine or until a process step is revisited if the process graph is cyclic. In the latter case, a cyclic state machine is obtained and it can happen that a state machine without end states is constructed. In this case, a warning is issued as this would mean that the business object is processed in an infinite loop that never terminates. Process steps, which are unordered with respect to each other, introduce multiple successor states and lead to nondeterministic state machines. There must at least be one edge whose start state is not the end state of another edge, i.e., there must be one clearly identifiable first activity in the process graph. This state is marked as the initial state of the state machine, otherwise the construction fails. Figure 8 shows the result of the construction. The state machines have the common symbols e and c, but each machine also has private symbols that the others do not read.

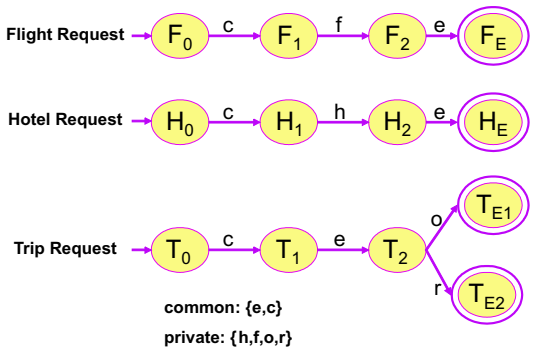

Figure 8. Three state machines for the discovered business objects.

5 Correctness of Model-Driven Transformations

The correctness of each of the model-driven transformations, which the TE implements, has to be shown separately. The basis for our correctness statements is the process graph that can be seen as a normalized, formal representation of a business view model. As the mapping of a particular business modeling approach to the process graph is essentially heuristic, given the informally specified semantics of common modeling approaches, we cannot make a precise formal argument about the relationship between the original business view model and the synthesized elements of the IT architectural model. However by assuming that our heuristic mapping is reasonable, we can carry over the correctness results in an informal way to the original input.

In the following, we sketch how the correctness of the BPEL4WS and ADoc synthesis can be established independently of each other. In a comprehensive transformation tool, sets of BPEL4WSs and ADocs would be generated as solution components of the IT architectural model. The architectural model will also contain other components and it has to determine how these components will interact, i.e., the component "wiring" should be made explicit in the model. A comprehensive correctness proof will then involve to consider each of the generated components as well as their modeled interaction, which goes far beyond the scope of this paper.

The BPEL4WS synthesis is correct if the synthesized BPEL4WS specification, when executed, generates mes-

sage exchange traces of partner interactions as they have been specified by the process graph in the following sense: Each subprocess vertex in the process graph is considered as representing a partner. A link from a subprocess vertex to a process-step vertex is interpreted as the receiving of some information from some partner, while a link from a process-step vertex to a subprocess vertex defines the sending of some information to the partner. The directed links between the process steps define a (partial) ordering of the sending and receiving activities in the process. The possible paths through the process steps thus define a set of possible conversation behaviors.

The structured activities in BPEL4WS (*switch, sequence, link*) also define a partial ordering of the atomic `<receive>` and `<invoke>` activities. A complete execution of the BPEL4WS specification, in which all branches are exhaustively enumerated and where all possible interleaving orderings for the unordered activities are determined, must generate exactly the set of possible conversation behaviors described by the process graph. Our transformation method is correct if the process subgraph containing only process steps and no subprocesses (i.e., only the white nodes from Figure 7) and their linking edges is acyclic. Otherwise, we would generated cyclic BPEL4WS specifications, which are not allowed. Note that the overall process graph containing subprocesses, which are mapped to BPEL4WS partners, can contain cycles.

The correctness criterion for the state machine synthesis is based on the fundamental relationship between regular languages and nondeterministic automata.

Take a process graph $G = (V, E, T)$ with process steps V and typed, directed links E between the process steps. We consider the names of the vertices the symbols of a language $L(G)$. The directed links define an ordering between the language symbols. We can formulate words over the language $L(G)$, but we are only interested in the sublanguage $L^-(G)$ of words defined through the paths of the graph G. Now take a set of ADoc state machines that we have constructed for this process graph.

Definition 6 *A nondeterministic finite state machine is a tuple $\langle S, s_0, E, I, \delta \rangle$ where*

- *S is a finite set of states,*
- *$s_0 \in S$ is the start state,*
- *$E \subseteq S$ is the set of end states,*
- *I is the input alphabet,*
- *$\delta : S \times I \to 2^S$ is the transition function*

Informally, a machine M accepts a word $w \in L$ if it transits through a sequence of states $s_0, s_1, \ldots, s_E$ when "reading" the word. The states of the state machines describe the possible states of the objects. A transition of a state machine takes a state s and an input symbol a and yields a new state s'. This means, when being in state s, the machine M reads symbol a and transits into state s'. Alternatively, we can interpret this as follows: when object $o \in O$ is in state s, the machine executes activity a and (upon completion of a) it enters state s'. This way, we can define the language $L(M)$, which is accepted by the machine M. Each of the state machines $M_1, \ldots, M_n$ that we constructed for the objects $o_1, \ldots, o_n$ has its own input alphabet $I_1, \ldots I_n$ determined from the activity names, which process the object in the subflow graph. We can consider the tuple $(I_1, \ldots, I_n)$ a distributed alphabet, which specifies the activities a machine can perform. When two different input alphabets share a symbol a, i.e., $a \in I_i \cap I_j$ with $i \neq j$, then a is called a *synchronization activity*. When a machine is in a state s that enables $s \xrightarrow{a} s'$ as its only transition, then this machine has to wait until all other machines that have a in their input set are ready to read a.

The behavior of the combined machines can be described with the synchronized shuffle product [16], which is defined as follows:

Definition 7 *Let $M_1 = \langle S_1, s_{01}, E_1, I_1, \delta_1 \rangle$ and $M_2 = \langle S_2, s_{02}, E_2, I_2, \delta_2 \rangle$ be two finite state machines. The shuffle product $M = M_1 \otimes M_2 = \langle S, s_0, E, I, \delta \rangle$ is defined as*

- $S = S_1 \times S_2$
- $s_0 = (s_{01}, s_{02})$
- $E = E_1 \times E_2$
- $I = I_1 \cup I_2$
- $\delta : (S_1 \times S_2) \times I \to 2^{(S_1 \times S_2)}$

with

$$\delta((s_1, s_2), i) = \begin{cases} (\delta_1(s_1, i), \delta_2(s_2, i)) & \textit{if} \quad i \in I_1 \cap I_2 \\ (\delta_1(s_1, i), s_2) & \textit{if} \quad i \in I_1 \setminus I_2 \\ (s_1, \delta_2(s_2, i) & \textit{if} \quad i \in I_2 \setminus I_1 \\ \textit{undefined} & \textit{otherwise.} \end{cases}$$

The shuffle product of state machines is commutative, i.e., we can take our object state machines and shuffle them in any order to build the shuffle machine $M = \bigotimes_i M_i$. This machine accepts language L(M). With that, we arrive at the criterion that our state machine synthesis needs to satisfy

$$\boxed{L^-(G) = L(M)}$$

meaning the object state machines together recognize the words of the language defined by the process graph. In other

words, the object state machines can only invoke those activity sequences that have been defined by the business view model.

Figure 9 shows the reachable states of the shuffle product machine that combines the three object state machines. The machine has $4 \times 4 \times 5 = 80$ states, but only 8 of them are reachable. We can see that the machines synchronize on the c and e activities, whereas the h and f activities can be executed in any order. One can also observe that the shuffle product machine, when executed, will perfectly generate the workflows one would normally implement for our example process graph, i.e., there is no need to additionally specify these workflows.

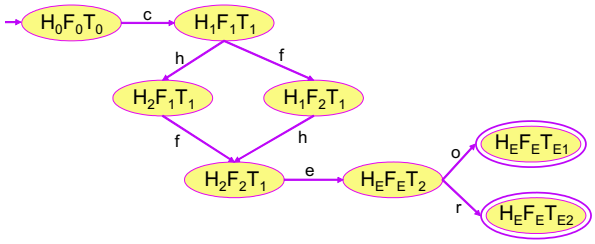

Figure 9. Synchronized shuffle product of the object state machines.

Cyclic or nondeterministic transitions can be handled with no problem and they inherit the usual semantics from automata theory. The only restriction that we currently establish is the existence of a unique initial state and of at least one end state in each state machine. The formalism also directly applies to I/O automata models where we additionally define a set of output symbols and an output function, which either depends on the state only (for a Moore-type machine) or on the state and the input symbol (for a Mealy-type machine), see for example [13]. The expressivity of these extended machine models is the same as that of the standard model, but they can yield more compact representations.

We can always transform any process graph into a set of state machines whose shuffle product will accept the language defined by the graph. The correctness criterion that we based on the equivalence of two regular languages is decidable. For deterministic automata, decision algorithms of polynomial complexity exist, but in the case of nondeterministic automata, the decision problem is NP-hard. Whether our current constructive algorithm can guarantee that a set of correct state machines is constructed is currently an open question that we need to investigate further.

6 Related Work

Our work relates to many different fields in computer science. The OMG initiative of model-driven architectures has motivated several approaches that address the problem of model transformation. A good example of this work is [11], which—like us—also aims at supporting the complete development cycle. In this approach, the focus lies on the definition of transformation schemes in UML, i.e., on an explicit representation of how the objects used in one model representation are mapped to the objects used in another model representation. As long as object-to-object mappings are sufficient, explicit transformation schemes can be defined in a straightforward manner. However, when a more complex algorithmic analysis has to be performed before a transformation can take place, transformation schemes easily reach their limits as the authors of [11] assert: *"The details of a transformation are only specified in terms of an unspecified procedural expression . . . "*.

In [12], a rule-based executable language to describe model transformations is proposed that allows programmers to write arbitrarily complex transformation programs. The expressivity and generality of the approach constitutes also its main drawback—it is up to the programmer to make sure that his program performs correct transformations and that the underlying rule set is consistent. No tool support for consistency checking is available, nor is it known whether the consistency of a given rule set is decidable.[2]

The work on transformations within the Kent Modeling Framework [1, 6] comes very close to our approach. In [6], mappings between a specification model and a design model are investigated. Both models are represented using a subset of UML (in particular class diagrams and state machines) and operations in the specification model are mapped to sequences of states in the design model. The authors discuss the necessity to formalize the mapping and to prove its correctness, but cannot yet provide a solution to these difficult problems. A very interesting contribution is the set of benchmarks that is proposed to evaluate mapping approaches and that we will consider in our ongoing work. In [1], a representation of mappings based on mathematical relations is proposed that is very suitable when the models are given in the form of classes, i.e., when sets of objects have to be related to each other.

Our IT architectural models are based on process communication protocols (represented in BPEL4WS) and communicating finite-state machines. This makes our work related to the fields of protocol conversion [7], control theory and the synthesis of controllers, see [13] for an example, and to the numerous approaches that model the behavior of discrete systems or software programs based on the theory of formal languages and automata. The synchronized shuffle product of automata, which we used to formally define a correctness criterion for our transformation method, is a

[2] An OMG standard is currently emerging that addresses the problem of representing queries, views, and transformations on UML2 (MOF) models, which will also become relevant to our future work.

recognized operation to describe sequentialized execution histories of concurrent processes. Using automata theory as the formal basis of our work has several advantages: we obtain automatic tools with well-defined properties, the dynamic behavior of modeled processes can be precisely defined, and many interesting properties of our models can be verified.

7 Summary and Outlook

In this paper, we investigate model-driven transformations that map business view models into IT architectural models and vice versa. We analyze two approaches to business process modeling, ADF and UML2 activity models, and define a mapping of these models to so-called *process graphs*, which possess a formal semantics and represent the structure of a process. We use graph-theoretic methods to analyze process graphs and to decompose them into *structured information flows*. These flows are used to automatically synthesize business protocol specifications and adaptive documents whose behavior is controlled by finite-state machines.

Our methods would also carry over to more elaborate process models that capture the *dynamic* behavior of processes, but unfortunately, many business process modeling approaches are not yet precise enough in their semantics when it comes to the representation of dynamic behaviors. Although the OMG defines a model as *"a formal specification of the function, structure and/or behavior of a system"* [9], formality in the sense of mathematical rigor is what is the hardest to achieve in most of the modeling approaches. The UML2 specification even states that *"Currently, the semantics are not considered essential for the development of tools, however this will probably change in the future."*, see [18], page 2-13. Our paper clearly shows what can be achieved in model-driven transformations when state-of-the-art modeling approaches are used with care. It also shows how essential semantics is for tools that try to do more than editing and syntax checking of models and how much more would become achievable if the semantics of dynamic processes— the spawning of subprocesses and their synchronization, the creation and destruction of objects at runtime—would be formally captured. In this case, a truly semantics-preserving transformation between models based on our methods is within reach.

References

[1] D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In *5th International Conference on UML*, volume 2460, pages 243–258. Springer, 2002.

[2] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. The web services description language WSDL. http://www-4.ibm.com/ software/ solutions/ webservices/ resources.html, 2001.

[3] F. Curbera et al. Business process execution language for web services. www-106.ibm.com/developerworks/webservices/library/ws-bpel/, 2002.

[4] E. Deborin et al. *Continuous Business Process Management with HOLOSOFX BPM Suite and IBM MQSeries Workflow*. IBM Redbooks, 2002.

[5] *Proceedings of the 6th International Enterprise Distributed Object Computing Conference*. IEEE Press, 2002.

[6] S. Kent and R. Smith. The bidirectional mapping problem. *Electronic Notes in Theoretical Computer Science*, 82(7), 2003.

[7] S. Lam. Protocol conversion. *IEEE Transactions on Software Engineering*, 14(3):353–362, 1988.

[8] F. Leymann and D. Roller. *Production Workflow*. Prentice Hall, 2000.

[9] Model-driven architecture - a technical perspective. Document from the Architecture Board MDA Drafting Team, 2001.

[10] P. Nandi, S. Kumaran, T. Heath, K. Bhaskaran, and R. Dias. ADoc-oriented programming. In *Proceedings of the International Symposium on Applications and the Internet (SAINT-2003)*. IEEE Press, 2003.

[11] J. Oldevik, A. Solberg, B. Elvesaeter, and A. Berre. Framework for model transformation and code generation. In EDOC-03 [5], pages 181–189.

[12] M. Peltier. MTrans, a DSL for model transformation. In EDOC-03 [5], pages 190–199.

[13] A. Ramirez, C. Sriskandarajah, and B. Behabib. Control of flexible-manufacturing workcells using extended moore automata. In *IEEE International Conference on Robotics and Automation*, pages 120–125, 1999.

[14] N. Sato, S. Saito, and K. Mitsui. Optimizing composite webservices through paralleliation of service invocations. In EDOC-03 [5], pages 305–316.

[15] A. W. Scheer, F. Abolhassan, W. Jost, and M. Kirchner. *Business Process Excellence - ARIS in Practice*. Springer, 2002.

[16] A. Shaw. Software descriptions with flow expressions. *IEEE Transactions on Software Engineering*, 4(3):242–254, 1978.

[17] Unified modeling language superstructure, version 2.0. 2nd revised submission to OMG RFP ad/00-09-02, 2003.

[18] Unified modeling language infrastructure, version 2.0. 2nd revised submission to OMG RFP ad/00-09-01S, 2003.