Model-Driven Development of Adaptive Service-Based Systems with Aspects and Rules

Jian Yu, Quan Z. Sheng, and Joshua K.Y. Swee

School of Computer Science
The University of Adelaide, SA 5005, Australia
{jian.yu01,qsheng,kheng.swee}@adelaide.edu.au

Abstract. Service-oriented computing (SOC) has become a dominant paradigm in developing distributed Web-based software systems. Besides the benefits such as interoperability and flexibility brought by SOC, modern service-based software systems are frequently required to be highly adaptable in order to cope with rapid changes and evolution of business goals, requirements, as well as physical context in a dynamic business environment. Unfortunately, adaptive systems are still difficult to build due to its high complexity. In this paper, we propose a novel approach called MoDAR to support the development of dynamically adaptive service-based systems (DASS). Especially in this approach, we first model the functionality of a system by two constituent parts: i) a stable part called the *base model* described using business processes, and ii) a volatile part called the *variable model* described using business rules. This model reflects the fact that business processes and rules are two significant and complementary aspects of business requirements, and business rules are usually much more volatile than business processes. We then use an aspect-oriented approach to weave the base model and variable model together so that they can evolve independently without interfering with each other. A model-driven platform has been implemented to support the development lifecycle of a DASS from specification, design, to deployment and execution. Systems developed with the MoDAR platform are running on top of a BPEL process engine and a Drools rule engine. Experimentation shows that our approach brings high adaptability and maintainability to service-based systems with reasonable performance overhead.

Keywords: Adaptive systems, Web service, aspect-oriented methodology, model-driven development, business processes, business rules.

1 Introduction

The service-oriented computing (SOC) paradigm, which promotes the idea of assembling autonomous and platform-independent services in a loosely coupled manner to create flexible business processes and applications that span organizational boundaries, presently has a dominant position in developing distributed, especially Web-based, software systems [10,16]. Web services are the most promising technology for implementing SOC, and Web Services Business Process Execution Language (WS-BPEL, or BPEL in short) [19] has become a de facto industry standard to create composite service processes and applications.

One of the key research challenges in SOC is *dynamically adaptive processes*. As stated in [16], "*services and processes should equip themselves with adaptive service capabilities so that they can continually morph themselves to respond to environmental demands and changes without compromising operational and financial efficiencies*". By *dynamically adaptive*, we mean a process is able to change its behavior at runtime in accordance with the changes in the external environment. To be specific, environmental changes could be classified into *execution* environment changes and *requirements* environment changes. Execution environment changes often demand an adaptive service-based system (DASS) to reconfigure itself in order to keep up with the *prescribed* quality-of-service (QoS) requirements. For example, a DASS may dynamically increase the number of service instances if its current response time is slower than the required criterion. On the other hand, requirements environment changes raise *new* requirements to a DASS. The requirements changes could be either *non-functional*, which also lead to system reconfiguration, or *functional*, which requires a DASS to include/exclude/replace component services or change its internal structure to expose a new functional behavior. For example, a business promotion campaign may require a travel booking process to include free car rental services or give discount to specific itineraries.

In this paper, we propose a novel approach called MoDAR (Model-Driven Development of DASS with Aspects and Rules) to facilitate the development of DASS that are able to dynamically adapt their behaviors in line with functional requirement changes. Based on the three-layer abstraction of service-based systems [15], at the top business process management (BPM) layer, we use business processes to model the relatively stable part of a DASS, which is called the *base model*, and use business rules, which are much easier to change than business processes [18], to model the volatile part, which is called the *variable model*. Such separation of concerns is crucial to manage the complexity of a DASS. An aspect-oriented approach is used to integrate business processes and rules so that they can keep their modularity and evolve independently. At the middle service composition and coordination layer, we use BPEL to represent business processes and encapsulate business rules as Web services. The invocations from a BPEL process to rule services are established based on the aspect model defined at the BPM layer. At the bottom service infrastructure layer, a BPEL engine and a Drools[1] rule engine work together to execute the DASS. A model-driven approach is adopted to automate the model transformation between layers. A service-based system created using MoDAR is dynamically adaptive in the sense that any changes made to the variable model of the system can be reflected dynamically in the running system.

Except for bringing dynamic adaptivity to service-based systems, another innovative feature of MoDAR is to use a model-driven approach to facilitate the

[1] www.jboss.org/drools/

modeling of DASS at different layers and the (semi-)automatic translation from higher-level models to executable systems, which eases both development and maintenance of DASS. A graphical model-driven platform has been developed to support the modeling, transformation, and deployment of DASS, which significantly reduces the system development time and cost.

The rest of the paper is organized as follows. Section 2 briefly overviews some basic concepts used in the paper and gives a motivating scenario that will be referred throughout the paper to illustrate our approach. Section 3 is dedicated to the MoDAR methodology. We first overview the whole approach and then explain in detail the three major layers of MoDAR and the model transformation between layers. Section 4 introduces the MoDAR platform and its performance evaluation. Finally, we discuss related work in Section 5 and conclude the paper in Section 6.

2 Background

In this section, we first briefly overview some basic concepts, namely *business rules*, *aspect-oriented methodology*, and *model-driven development*. We then present a motivating scenario to highlight the challenges in developing DASS.

2.1 Business Rules

Business rules are statements that define or constrain some aspects of a business [12]. They make the knowledge about *regulations*, *policies*, and *decisions* explicit and traceable [19]. Business rules can be classified into four groups:

- *constraint rule*: A statement that expresses an unconditional circumstance that must be true or false.
- *action enabler rule*: A statement that checks conditions and initiates some actions upon finding the conditions true.
- *computation rule*: A statement that checks a condition and when the result is true, provides an algorithm to calculate the value of a term.
- *inference rule*: A statement that tests conditions and establishes the truth of a new fact upon finding the conditions true.

2.2 Aspect-Oriented Methodology

Aspect-oriented methodology (AOM) is a strand of software development paradigm that models scattered crosscutting system concerns as first-class elements, and then weaves them together into a coherent model or program. Concerns can be high-level notations like security and quality of service, low-level notations like caching and buffering, functional elements such as features or business rules, or non-functional elements such as synchronization and transaction management [8]. AOM can bring better modularity: it allows developers to

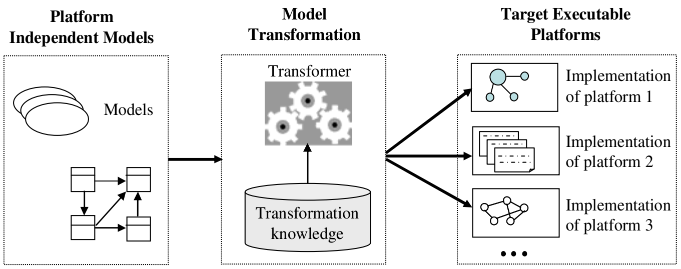

Platform Independent Models
Models
Model Transformation
Transformer
Transformation knowledge
Target Executable Platforms
Implementation of platform 1
Implementation of platform 2
Implementation of platform 3
•••

Fig. 1. Model-driven development

encapsulate system behavior that does not fit cleanly into the particular programming model in use [3]. Referring to Aspect4J [11], an aspect-oriented program usually consists of base modules and a number of aspects that modularizes crosscutting concerns. Basic concepts in Aspect4J include *aspect*, *pointcut*, and *advice*. An aspect wraps up pointcuts and advices. A pointcut picks out certain join points, which are well-defined points (e.g., method calls) in the program flow. An advice is a piece of code that is executed when a join point is reached. There are three types of advice:

– A *before advice* runs just before the join points picked out by the pointcut.
– An *after advice* runs just after each join point picked out by the pointcut.
– An *around advice* runs instead of the picked join point.

2.3 Model-Driven Development

Model-driven development (MDD) [9] is an approach that supports system development by employing a model-centric and generative development process. The basic idea of MDD is illustrated in Figure 1. Adopting a higher-level of abstraction, software systems can be specified in platform independent models (PIMs), which are then (semi)automatically transformed into platform specific models (PSMs) of target executable platforms using some transformation tools. The same PIM can be transformed into different executable platforms (i.e., multiple PSMs), thus considerably simplifying software development.

2.4 A Motivating Scenario

In this section, we present a context-aware travel planning scenario in the mobile commerce domain to motivate the challenges in developing DASS.

Suppose that a travel agency StarTravel wants to provide a mobile application called SmartTravel to its customers, as depicted in Figure 2. With SmartTravel, a customer can send travel requests to StarTravel via her mobile phone. When receiving a request, SmartTravel first validates the request, e.g., to see if the request contains valid departure and destination cities, and if the validation fails, the customer will receive an SMS explaining why the request is failed. Otherwise,

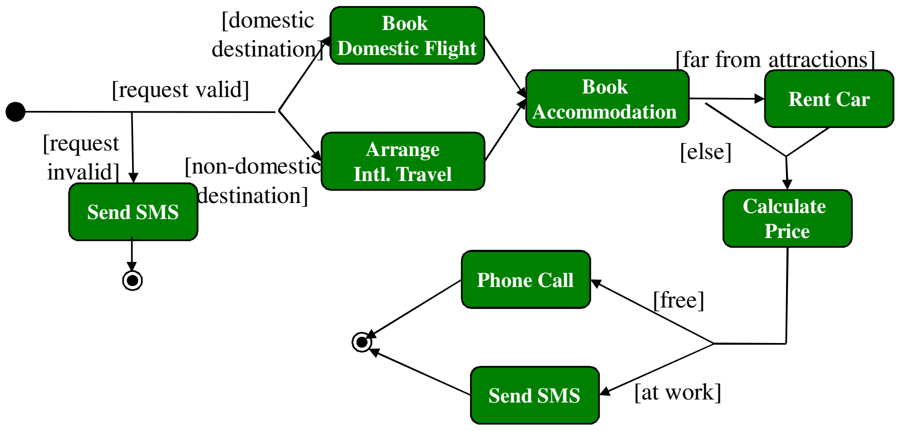


Fig. 2. The SmartTravel business process

based on the destination of the travel request, SmartTravel will either book a domestic flight or arrange an international travel. SmartTravel will also book the accommodation and rent a car for the customer based on her preferences. Finally, the customer could get a 10% discount if she is a member of StarTravel and the detailed travel plan will be sent back to the customer based on her presence: if she is free, a staff will give her a call; and if she is at work, an SMS will be sent instead.

The constant change in business environment, organization policies, and also user preferences and context demand the business logic of SmartTravel to evolve dynamically to be in line with the new requirements. For example, following requirements may crop up at anytime in the future:

– Booking domestic flight and arranging international travel are merged to a single service because of company policy adjustment.
– The customer wants to rent a car only if the location of her accommodation is far from tour attractions.
– If there is a promotion, the cost needs to be re-calculated using the promotion rule.
– The customer wants to receive the travel plan via email when she is in a meeting.

A major challenge is how to make SmartTravel dynamically adaptive so that when a new requirement emerges it is able to promptly adjust its behavior with minimal effort. If we follow the common SOC approach to implement Smart-Travel as an executable BPEL process, the process structure may need to change in many new situations, which results in *re-compilation* and *re-deployment* of the process. For example, if the business policy has changed and domestic and international travels need to be merged into one service, we have to review the whole process logic, and then create a new service to accommodate the functions of

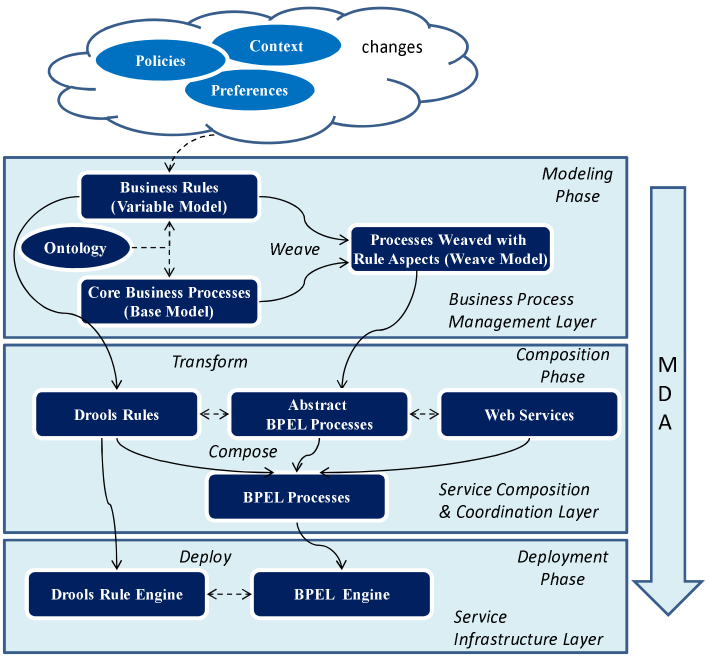

Policies
Context
Preferences
changes
Modeling Phase
Business Rules (Variable Model)
Ontology
Core Business Processes (Base Model)
Weave
Processes Weaved with Rule Aspects (Weave Model)
Business Process Management Layer
Transform
Composition Phase
Drools Rules
Abstract BPEL Processes
Web Services
Compose
BPEL Processes
Service Composition & Coordination Layer
Deploy
Deployment Phase
Drools Rule Engine
BPEL Engine
Service Infrastructure Layer
MDA

Fig. 3. MoDAR Overview

both services, and then adjust the process structure. Another challenge is that if we encode all parts of the requirements into business processes, the business rules in the requirements are hard-coded into the processes and their modularity and traceability are lost: there is no explicit association between a business rule and the process segment that implements this business rules, and if we change a business rule, the whole process needs to be changed.

3 The MoDAR Approach

3.1 Overview

In this section, we briefly introduce the MoDAR approach and discuss several key principles used in the design of this approach.

As illustrated in Figure 3, the MoDAR approach consists of three main phases, namely the *modeling phase*, the *composition phase*, and the *deployment phase*, based on the three-layer abstraction of service-based systems [15], and a model-driven approach is used to transform models between different phases.

Since DASS are generally difficult to specify, build, and validate due to its high complexity and variability [21,14], in the modeling phase, we adopt the basic principle of *separation of concerns* to manage complexity and variability: a system model is divided into a base model and a variable model. The base model represents the stable and constant part of a system and the variable model represents the volatile part. Specifically, business processes are used to specify the base model and business rules are used to specify the variable model. Business processes and business rules complement each other in a way that business processes capture the *flow* logic while business rules capture the *decision* aspect of a requirement. The reason that we use business rules to specify the variable model is that business rules are much easier to change than business processes [18,19,5]. As stated in [18], "*the most significant changes do not come from re-engineering workflow, but from rethinking rules*". Furthermore, a business process even becomes more stable and reusable if we manage to abstract its complex branch structures into rules [19].

To make the base model and the variable model semantically interoperable, we use a minimum set of ontology concepts as the basic elements in defining activity parameters in processes and also in defining rule entities. To minimize the effort of building a domain ontology, only named classes are used to construct classes and any other class constructors such as *intersectionOf*, *unionOf*, and *someValueFrom* are not used in MoDAR.

To make the base model and the variable model work together, we adopt an aspect-oriented approach to integrate them into a weave model. This approach ensures the modularity of the base model and the variable model so that they can evolve independently. If we directly translate rules into process structures and insert them into a process, both modularity and traceability of the rules are lost. In MoDAR, we can specify cutting points on a process where rules can be woven in. The three models will be described in details in Section 3.2.

In the composition phase, the variable model is translated into Drools rules and the weave model is transformed into an abstract BPEL process. In this abstract process, at every join point, the invocation to a rule aspect is translated to a Web service invocation. The process is abstract in a way that all the service invocations translated from the activities in the based model are still abstract. After all the abstract services in the process are associated with concrete Web services to implement their functionalities, the process then can be transformed into an executable BPEL process.

Finally in the deployment phase, the BPEL process and the Drools rules are deployed to their corresponding engines. Dynamic adaptivity is achieved in a way that as long as the interface between a rule and the process does not change, we can freely change the rule in the modeling phase and then translate and redeploy the rule without even stop the execution of the process.

3.2 The MoDAR Models

In this section, we discuss in detail how to create the base model and the variable model, and how to weave these two models using the running example in Section 2.

The Base Model. The base model represents the stable part of business requirements. In MoDAR, a base model is a simplified business process that can be defined by the tuple $< \mathcal{T}, t_s, t_e, Seq : \mathcal{T} \times \mathcal{T}, Par : \mathcal{T} \times \mathcal{T} >$, where $\mathcal{T}$ is the set of business activities, $t_s \in \mathcal{T}$ is the start event, $t_e \in \mathcal{T}$ is the end event, Seq is the sequential flow between two activities, and Par is the parallel flow between two activities. So we remove all the complex process control constructs that are related to conditional decisions, which are usually the volatile part of a process.

We define a business activity as a trinary tuple of *name*, *inputs*, and *outputs*: $t =< name, I : Name \times \mathcal{C}, \mathcal{O} : Name \times \mathcal{C} >$, where $Name$ is a finite set of names, $name \in Name$, and every input or output of a business activity has a name and a type/class, and the type must be a concept in an ontology: $\mathcal{C} \in Concept_{onto}$. The purpose that we associate an I/O parameter with an ontology concept is twofold: first, the ontology serves as the common ground between the base model and the variable model and thus makes these two models semantically interoperable; second, the semantics attached to business activities later can be used to semantically discover services that implement business activities.

To create a base model, we can start from scratch to describe the general steps of a business process and then connect these steps using sequential or parallel flow. For example, the general steps of the SmartTravel process can be illustrated by Figure 4.

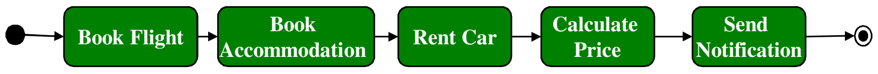


Fig. 4. The SmartTravel Base Model

Alternatively, we can also re-engineering legacy business processes to a base model using the concept of *variation point* ($\mathcal{VP}$). A $\mathcal{VP}$ includes any conditional structure (which is called *conditional* $\mathcal{VP}$) or activities that tend to change (which is called *activity* $\mathcal{VP}$) in a process. For example, in Figure 5, all the conditional structures have been identified as conditional $\mathcal{VP}$ and the `CalculatePrice` activity has been identified as an activity $\mathcal{VP}$.

For conditional $\mathcal{VP}$, we have two strategies to convert them to sequential flow:

- If this $\mathcal{VP}$ is served as a precondition/constraint for further steps of the process (which means the other branch terminates the process), we replace this $\mathcal{VP}$ with a sequence flow. Later on, a rule will be created based on this $\mathcal{VP}$ and weaved to the process. For example, in Figure 5, we can apply this strategy on $\mathcal{VP}_1$, which is used to check the validity of requests.
- If the $\mathcal{VP}$ is served as a branch for alternative activities, which means different branches will eventually merge, we use an abstract activity to replace this branch structure. Later on, a rule will be created to do the actual activity based on the $\mathcal{VP}$ conditions. For example, in Figure 5, we can apply this strategy to $\mathcal{VP}_2$, $\mathcal{VP}_3$, and $\mathcal{VP}_5$.

For an activity $\mathcal{VP}$, we keep it intact but later on we will weave rules onto it to adjust its behavior. By identifying $\mathcal{VP}$ and applying the above strategies, we can convert the SmartTravel process (Figure 2) to a MoDAR base model as shown in Figure 4.

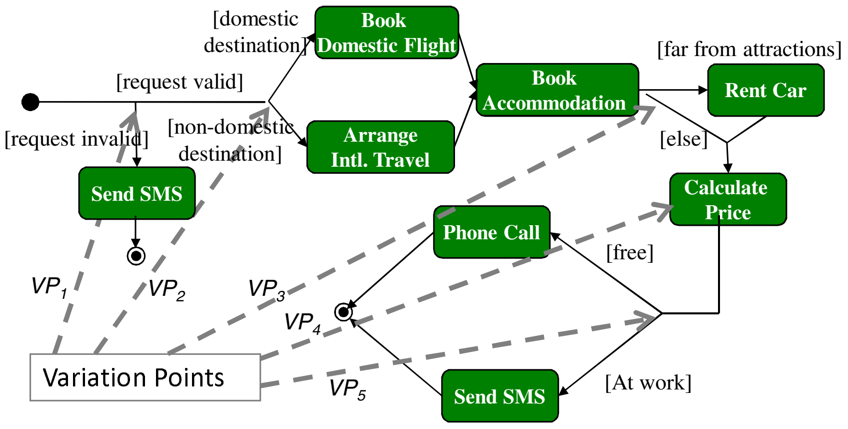


Fig. 5. Variation Points in the SmartTravel Business Process

The Variable Model. A variable model consists of a set of rules where each rule is defined by a binary tuple: $r =< condition, action >$. The *action* is enabled only if the *condition* is true. For any variable v in a condition or action expression, its type must be a concept in the same ontology used by the base model. If the type of a variable can uniquely identify a variable, we do not explicitly define this variable but use its type to represent it. An action expression can be any numeric or string expressions. For example, $Price * 10\%$, $FirstName +' .' + LastName$. To facilitate the definition of process related actions, we also define some standard actions with prescribed semantics:

- `[SkipActivity]`: skip an activity;
- `[SkipActivityThenAction]`: skip an activity then do an action. The action could be calling a service;
- `[Abort]`: abort the running of the process;
- `[ActionThenAbort]`: do an action before abort the running of the process.

The following are two rule examples:

If the departure airport or arrival airport of a travel request is empty, stop processing the request:
Condition: `DepartureAirport` equals to "" or `ArrivalAirport` equals to ""
Action: `[ActionThenAbort(SendSMS)]`

If a customer is a frequent flyer, give him a 5% discount:
Condition: `Customer has FrequentFlyer` equals to true
Action: `Price` equals to `Price` * 0.95

The first example is a constraint rule. The ontology concepts `DepartureAirport` and `ArrivalAirport` implicitly represents two variables, and the action is

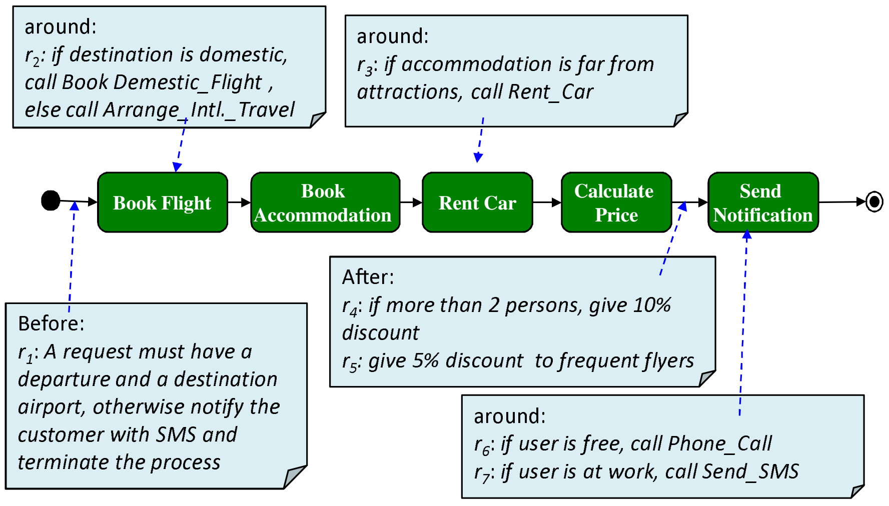

around:
r2: if destination is domestic, call Book Demestic_Flight , else call Arrange_Intl._Travel
around:
r3: if accommodation is far from attractions, call Rent_Car
Book Flight
Book Accommodation
Rent Car
Calculate Price
Send Notification
After:
r4: if more than 2 persons, give 10% discount
r5: give 5% discount to frequent flyers
Before:
r1: A request must have a departure and a destination airport, otherwise notify the customer with SMS and terminate the process
around:
r6: if user is free, call Phone_Call
r7: if user is at work, call Send_SMS

Fig. 6. The SmartTravel Weave Model

to send an SMS then abort the execution. The second example is a computation rule. In its condition, `Customer` is a complex class, and one of its property is `FrequentFlyer`.

The Weave Model. In MoDAR, an aspect weaves a rule to a process. It can be defined by: $a \in \{Before, Around, After\} \times \mathcal{T} \times \mathcal{R}$, where $\mathcal{R}$ is the set of rules. Similar to Aspect4J, we also identify three types of aspect: *before aspects*, *around aspects*, and *after aspects*. An aspect is always associated with a business activity, and it is activated either before, after the execution of the activity or replace the execution of the activity. From another perspective, $event \in \{Before, Around, After\} \times \mathcal{T}$ becomes the triggering event of a rule.

It is worth noting that the interoperability between an activity and its associated rules is established through the predefined ontology. For example, the input parameters of the `Book Flight` activity must contain two parameters having semantic annotation `DepartureAirport` and `ArrivalAirport`, so that the associated rule (see the first rule in the variable model) can use these two concepts in defining itself.

Figure 6 shows a weave model of the SmartTravel process, where the base model is weaved with rules from the variable model. As we can find in this weave model, all the requirement changes specified in Section 2.4 have been introduced by rules inside the aspects.

4 Implementation and Performance Evaluation

In this section, we discuss the implementation of the MoDAR platform, including how to transform models defined in the modeling phase (i.e., the base model, the variable model, and the weave model) to executable code. We also report some preliminary results on performance study.

The MoDAR platform contains three main components: the *Process Modeler* (the Modeler) for graphically modeling the base model, the rules, and also the weave model; the *Association and Transformation Tool* (AT Tool) for associating Web services with activities in the base model and actions in rules, and for generating and deploying executable code; and the *Business Domain Explorer* (Explorer) for exploring domain ontologies.

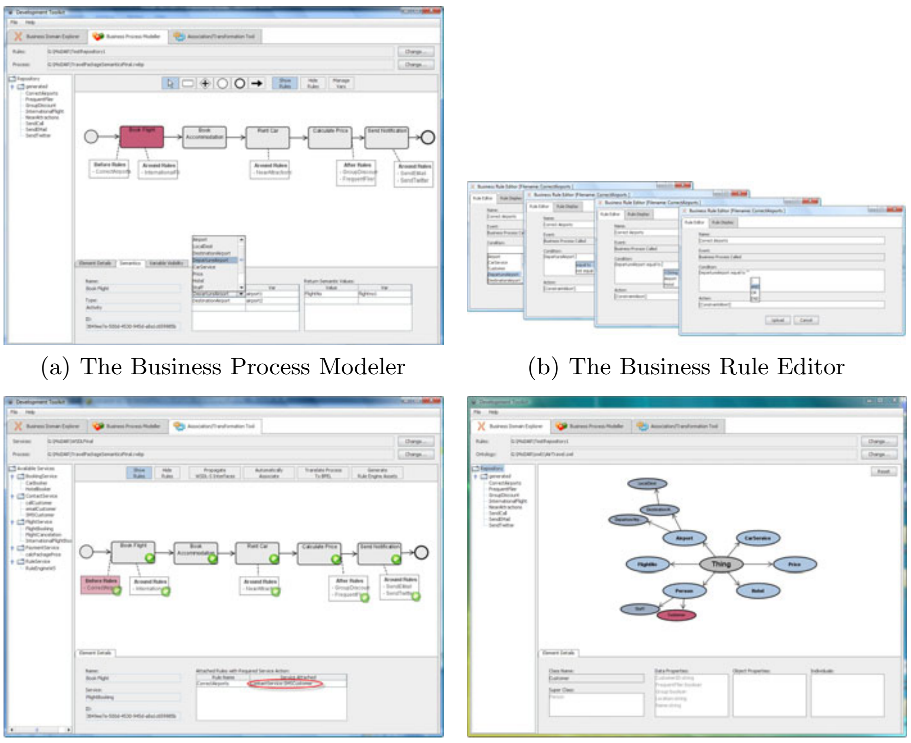
4.1 The Process Modeler

The Business Process Modeler provides a visual interface for defining the base model, the variable model, and also the weave model. As shown in Figure 7(a), the main canvas in the middle of the graphical interface of the Modeler is used to define and display the base model, and essential BPMN constructs including `Activity`, `Parallel Gateway`, `Start Event`, `End Event`, and `Sequence Flow` are displayed as a list of toolbar buttons on top of the canvas for visually creating the base model. If we click a specific activity in the base model, we can define its I/O semantics in the bottom pane. If we select a concept from the drop-down menu that contains all the concepts in the domain ontology as the type for a parameter, a variable is automatically created to represent this parameter. As shown in the snapshot, the `BookFlight` activity has two input parameters with type `DepartureAirport` and `DestinationAirport`, and one output parameter with type `FightNo`.

The middle canvas is also used to define the weave model. As shown in the snapshot, the left pane shows the list of business rules that can be drag-n-drop to an activity and becomes its *before/around/after rule*. It is worth noting that if a rule is associated with an activity, all the parameter variables of this activity are also visible to the rule.

A new business rule can be created in the left rule repository pane. As shown in Figure 7(b), the rule editor use the concepts in the domain ontology to define the *condition* and *action* components of a rule. To facilitate domain/application experts to create a correct business rule, the editor has a very nice auto-complete[2] feature: at every step in composing a condition or action, all the possible inputs are listed in a drop-down menu for the user to select. For example as shown in Figure 7(b), to write the constraint rule (the first one) as we discussed in Section 3.2, first we select the `DepartureAirport` concept from the drop-down menu, and then all the operators that are applicable to this concept are displayed in the follow-on menu, and so on. It is worth noting that all the I/O parameter variables in the based model that are visible to a rule will be automatically bind to the corresponding concepts in the rule, and if a rule needs to use variables other than the activity parameters it attached to, we can use the *variable visibility* tab in the bottom pane to make additional variable visible to this rule.

[2] `http://en.wikipedia.org/wiki/Autocomplete`



(a) The Business Process Modeler (b) The Business Rule Editor

(c) The Association and Transformation Tool (d) The Business Domain Explorer

Fig. 7. Main Components of the MoDAR Platform

4.2 The Association and Transformation Tool

The Association and Transformation Tool has two main functionalities. First, it provides a visual interface for associating Web services to process activities and rule actions. Second, it is used to automatically generate executable code and deployment scripts from the models defined in the modeling phase. It is worth noting that BPEL is selected as the targeting executable process language and Drools is selected as the targeting executable rule language.

Figure 7(c) shows the graphical interface of the AT Tool. The left pane shows a directory of Web services. The user can drag-n-drop a Web service to an activity or rule in the middle canvas. Since semantic service discovery and match is not the focus of our work, currently we only support a basic manual association method: a Web service can be associated with an activity/action if and only if they have the same number of inputs and outputs.

Transforming a weaved process model to a BPEL process is straightforward: BPMN constructs has a nice match with BPEL constructs, and invocation to an aspect is wrapped in a Web service call, either before, replace, or after calling the activity that the aspect attaches to. Figure 8 shows a BPEL code snippet

```
<!-- Book Flight Before Rules Service Execution-->
<bpws:invoke inputVariable ="rulerequest"
 name="RequestRuleService0"  operation="RuleEngineWS"
 outputVariable="ruleresponse" partnerLink="rule"
 portType="rls:RuleService"/>
<!-- Book Flight Before Rules Variable Propagation -->
<bpws:assign name="Propagate_BookFlightBeforeRules" validate="no">
...</bpws:assign>
<!-- Book Flight Before Rules Logic Copy -->
...
<!--  Book Flight Before Rules Logic Test - Abort Action -->
<bpws:if><bpws:condition>bpel:contains($boolTest, "true")
  </bpws:condition><bpws:sequence>
      <!-- Alternate Service Executed on Rule Engine -->
      <bpws:throw name="cancelProcess" faultName="cancelation"/>
</bpws:sequence></bpws:if>
```

Fig. 8. SmartTravel BPEL Code Snippet

that is generated from the weave model. In the code snippet, first the before rule service of the `Book Flight` service is invoked, and if the rule service returns `true`, which means abort the process, the execution of the process will be aborted by throwing a `cancelProcess` exception. It is worth noting that if the action part of a rule needs to invoke a Web service, the invocation is included in the wrapping Web service of the rule so that Drools engine is self-contained and does not need to have the capability to invoke Web services.

All the business rules are automatically translated to Drools rules. Finally, an *ant*[3] script is generated for deploying the BPEL code and the Drools rules code to the corresponding engines.

4.3 The Business Domain Explorer

Figure 7(d) is a snapshot of the Explore graphical interface. It displays the domain ontology that the SmartTravel application is based on. This ontology only contains the minimum concepts that are used to define the semantics for the I/O parameters of business activities and the variables in business rules. From this interface, the process designer can get familiar with the concepts in the domain. OWL [13] is used as the ontology language.

4.4 Performance Evaluation

This section reports some initial performance evaluation of the MoDAR platform. A PC with Intel Core i7 860 CPU and 4GB RAM is used as the test-bed server. JBPM-BPEL-1.1.1[4] running on top of JBOSS-Application-Server-4.2.2 is used as the BPEL engine and Drools-5.0[5] is used as the rule engine.

[3] `http://ant.apache.org/`
[4] `http://www.jboss.org/jbpm`
[5] `http://www.jboss.org/drools`

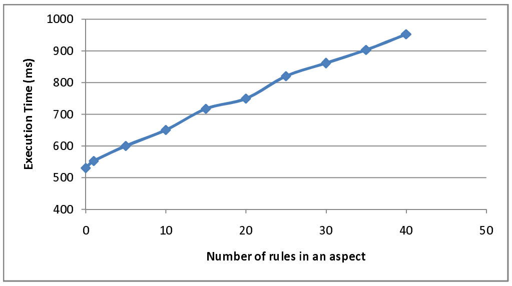


Fig. 9. Execution time of an aspect

The overhead of a weaved BPEL process mainly lies in its aspects, which are implemented as Web service invocations. The first experiment we have conducted is to test the performance impact of the number of rules in a single aspect. As illustrated in Figure 9, the execution time of an aspect increases steadily from around 552ms for one rule to 952ms for 40 rules.

Next we compare the execution time of two implementations of the running scenario: one is a manually composed BPEL process and the other is the weaved process generated by the MoDAR platform, with all the activity Web services deployed locally. Since the scenario includes some complex services such as `Book Flight` and `Book Accommodation` that need to communicate with external systems, on average, the pure BPEL process takes about 18.125s to execute and the weaved process takes about 18.715s. We can see that the overhead of the weaved process compared to the pure BPEL process is 590ms, or 3.3%.

From the above experiments, we can conclude that the MoDAR platform is suitable for dealing with processes that involve complex business logic. It dramatically simplifies the development process and makes the executable processes dynamically adaptable with a very small performance overhead.

5 Related Work

Dynamic and adaptive processes are one of the key research challenges in service-oriented computing [16]. Existing approaches can be classified into two main categories: adaptation to non-functional changes and adaptation to functional changes. While the former investigates the problem of how to adapt service configurations to satisfy multiple QoS requirements [20,2], the latter focuses on making the service-based systems adaptable to any business and environmental changes that require the system to change its functional behavior. Clearly, our work falls in the latter case.

In [5], the authors propose AO4BPEL, where BPEL is extended with aspect-oriented features: pointcuts can be defined on a process to weave aspects, and BPEL constructs such as <`SWITCH`> and <`ASSIGN`> are used to compose rule aspects. In [17], the authors propose a runtime environment where both a BPEL engine and a rule engine are connected to a service bus and BPEL invocation activities are intercepted so that rules can be applied either before or after a service invocation. In [6], the authors propose SCENE, a service composition execution environment that supports the execution of dynamic processes using rules to guide the binding of operations. While all these approaches focus on providing executable languages and environments to improve the adaptability of service-based systems, MoDAR is a model-driven approach that supports the development of service-based systems from high-level modeling to low-level code generation and execution. Furthermore, AO4BPEL does not support dynamic adaptation since all the rules are part of the process and thus cannot be managed and changed independently.

In the general research area of software architecture, model-driven development is also combined with aspect-oriented methodology to manage the vast number of possible runtime system configurations of dynamic adaptive systems. In [14], the authors put system configurations in aspects and weave them at a model level to reduce the number of artifacts needed to realize dynamic variability. In [7], the authors propose an aspect-oriented approach for implementing self-adaptive system on the Fractal platform [4]. They separate adaptation policies from the business logic and use low-level reconfiguration scripts to do the actual adaptation actions. While existing approaches in this area usually need the support of an adaptive middleware such as Fractal [4] and OSGI [1], our targeting platform is BPEL, the de facto industry standard language for service composition.

6 Conclusion

In this paper, we have presented MoDAR: a model-driven approach for developing dynamic and adaptive service-based systems using aspects and rules. We have introduced the methodology of how to separate the variable part from a stable process and model the variable part as business rules, and how to weave these rules into a base process. We have also introduced a platform that supports the model-driven development of rule-weaved BPEL processes. The targeting system is dynamically adaptive in the sense that rules as the variable part of a system can change at runtime without affecting the base process. Experimental results show that systems generated with MoDAR have a reasonable performance overhead while significantly simplifies the development process.

In the future, we plan to develop real-life service-based applications to further validate effectiveness of MoDAR and test its overall usability. Performance optimization is another focus of our future research work.

References

1. The OSGi Alliance. OSGi Service Platform Core Specification Release 4.1 (2007), `http://www.osgi.org/Specifications/`
2. Ardagna, D., Pernici, B.: Adaptive Service Composition in Flexible Processes. IEEE Transactions on Software Engineering 33(6), 369–384 (2007)

3. Baniassad, E.L.A., Clarke, S.: Theme: An Approach for Aspect-Oriented Analysis and Design. In: Proc. of the 26th International Conference on Software Engineering (ICSE 2004), pp. 158–167 (2004)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.: The FRACTAL Component Model and its Support in Java. Software Practice and Experience 36, 1257–1284 (2006)
5. Charfi, A., Mezini, M.: Hybrid Web Service Composition: Business Processes Meet Business Rules. In: Proc. of the 2nd International Conference on Service Oriented Computing (ICSOC 2004), vol. 30–38 (2004)
6. Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 191–202. Springer, Heidelberg (2006)
7. David, P., Ledoux, T.: An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 82–97. Springer, Heidelberg (2006)
8. Elrad, T., Filman, R.E., Bader, A.: Aspect-Oriented Programming: Introduction. Commun. ACM 44(10), 29–32 (2001)
9. Frankel, D.S.: Model Driven Architecture$^{\text{TM}}$: Applying MDA$^{\text{TM}}$ to Enterprise Computing. John Wiley & Sons, Chichester (2003)
10. Georgakopoulos, D., Papazoglou, M.P.: Service-Oriented Computing. The MIT Press, Cambridge (2008)
11. Gradecki, J.D., Lesiecki, N.: Mastering AspectJ: Aspect-Oriented Programming in Java. Wiley, Chichester (2003)
12. The Business Rules Group. Defining Business Rules, What Are They Really? (2001), `http://www.businessrulesgroup.org`
13. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Chen, B.H.C.: Composing Adaptive Software. IEEE Computer 37(7), 56–74 (2004)
14. Morin, B., Barais, O., Nain, G., Jezequel, J.M.: Taming Dynamically Adaptive Systems using Models and Aspects. In: Proc. of the 31st International Conference on Software Engineering (ICSE 2009), pp. 122–132 (2009)
15. Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M.P., Pohl, K.: A Journey to Highly Dynamic, Self-Adaptive Service-Based Applications. Automated Software Engineering 15(3-4), 313–341 (2008)
16. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. Computer 40(11), 38–45 (2007)
17. Rosenberg, F., Dustdar, S.: Usiness Rules Integration in BPEL - a Service-Oriented Approach. In: Proc. of the 7th IEEE International Conference on E-Commerce Technology, pp. 476–479 (2005)
18. Ross, R.G.: Principles of the Business Rules Approach. Addison-Wesley, Reading (2003)
19. Vanthienen, J., Goedertier, S.: How Business Rules Define Business Processes. Business Rules Journal 8(3) (2007)
20. Yau, S.S., Ye, N., Sarjoughian, H.S., Huang, D., Roontiva, A., Baydogan, M.G., Muqsith, M.A.: Toward Development of Adaptive Service-Based Software Systems. IEEE Transactions on Services Computing 2(3), 247–260 (2009)
21. Zhang, J., Cheng, B.H.C.: Model-Based Development of Dynamically Adaptive Software. In: Proc. of the 28th International Conference on Software Engineering (ICSE 2006), pp. 371–380 (2006)