Model-driven development of adaptive web service processes with aspects and rules

Jian Yu a,∗, Quan Z. Sheng b, Joshua K.Y. Swee b, Jun Han c, Chengfei Liu c, Talal H. Noor b

a School of Computer and Mathematical Sciences, Auckland University of Technology, Auckland, New Zealand
b School of Computer Science, The University of Adelaide, Adelaide, Australia
c Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Australia

A R T I C L E   I N F O

Keywords:
Web services
Adaptive systems
Model-driven development
Aspect-oriented methodology
Design tools and techniques

A B S T R A C T

Modern software systems are frequently required to be adaptive in order to cope with constant changes. Unfortunately, service-oriented systems built with WS-BPEL are still too rigid. In this paper, we propose a novel model-driven approach to supporting the development of dynamically adaptive WS-BPEL based systems. We model the system functionality with two distinct but highly correlated parts: a stable part called the base model describing the flow logic aspect and a volatile part called the variable model describing the decision logic aspect. We develop an aspect-oriented method to weave the base model and the variable model together so that runtime changes can be applied to the variable model without affecting the base model. A model-driven platform has been implemented to support the development of adaptive WS-BPEL processes. In-lab experiments show that our approach has low performance overhead. A real-life case study also validates the applicability of our approach.

© 2014 Elsevier Inc. All rights reserved.

1. Introduction

Service-Oriented Computing is the computing paradigm that promotes the idea of assembling autonomous and platform-independent software services in a loosely coupled manner for creating business processes and applications that span organizational boundaries. It presently has a dominant position in developing distributed software systems [1–4]. Web services are one of the major technologies for implementing Service-Oriented Computing, and Web Services Business Process Execution Language (WS-BPEL, or BPEL in short) [5,6] has become a de facto industry standard (which has been widely adopted by major IT service providers including IBM, Oracle, and SAP) to create composite service processes and applications.

One of the key research challenges in developing service-oriented applications is dynamically adaptive processes. As stated in [2], “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 being able to change its behavior at runtime in accordance with the changes in the requirements and/or the external environment.
In general, changes in requirements or environment require a system to use different adaptation strategies to deal with them. One type of adaptation is non-functional, which means the system may need to adjust its scheduling algorithms or resources, or replace incompetent components in order to achieve the prescribed or new Quality-of-Service (QoS) attributes. Adaptation approaches proposed in [7] and [4] are examples of non-functional adaptation. The other type of system adaptation is functional, which requires an adaptive system to include/exclude/replace component services or change its processing logic to expose a new functional behavior. For example, a business promotion campaign may require a travel booking system to include free car rental services or give discount to specific itineraries. The merge/division in organizational structure may require the supporting software system to merge/divide its component services. The focus of this paper is on functional adaptation.

Although BPEL has been around for one decade (submitted to OASIS for standardization in April 2003), it is inherently static and has limited dynamic adaptation features [4]. BPEL supports dynamic binding of component services through partner links. However, the process definition itself would contain a large amount of code that is not related to the business process. Since then, a number of approaches (e.g., AO4BPEL [8] and VieDAME [4]) have been proposed to cope with this issue. Unfortunately, most of them are implementation level techniques that apply at the code development and execution stage.

From the software design perspective, we see the root of BPEL’s incapability in dealing with dynamic adaptation as a problem caused by software developers trying to satisfy requirements using only the process-oriented solution. Although human problem solving is innately procedural [9], and business processes are the main tool for capturing procedures or flow logic in business requirements, there are also business rules, policies, and regulations1 in the requirements which represent the decision aspect, and they usually tend to change than procedures. Just as Ross [10] stated, “the most significant changes do not come from re-engineering workflow, but from rethinking rules”. If we still use a process language to capture business rules, they are translated to imperative conditionals and branches in the process. Such an approach brings two major problems: i) any change to business rules requires the whole process to change, and ii) the traceability of business rules is usually lost in the translation.

In this paper, we propose a novel approach called MoDAR (Model-Driven Development of Dynamically Adaptive Service-Oriented Systems with Aspects and Rules) to support the systematic development of dynamically adaptive BPEL-based service-oriented systems. MoDAR adopts Model-Driven Development methodology (MDD) [11], in which software systems abstraction is specified in platform independent models (PIMs), and then the PIMs are (semi)automatically transformed into platform specific models (PSMs) using dedicated transformation tools. The MoDAR PIM is composed of three models: a base model, a variable model, and a weave model. The base model captures the flow logic of the requirements. The variable model captures the decision logic of the requirements. The weave model weaves the base model and variable model together using an aspect-oriented mechanism. The MoDAR PSM integrates BPEL processes and Drools2 rules, which is dynamically adaptive in the sense that rules, representing the decision logic, can be changed at runtime without redeploying the BPEL process. Along with the methodology, we have developed an intuitive graphical development environment and a prototype execution environment. A real-life case study in the health service domain has been conducted to demonstrate the applicability of our approach. Performance evaluation on the execution environment was also conducted to demonstrate its efficiency.

A preliminary result of this study has been published in [12]. This article is a comprehensive summary of the MoDAR project and adds the following original major contributions:

- A formally defined base model language and its variable activity concept that supersedes the previous immature variable point concept,
- A formally defined variable model with an ontology-based rule language,
- A detailed report of our substantial new development to the MoDAR platform, and
- An extensive evaluation of the MoDAR approach, including a performance study, a real life case study in the healthcare industry, and an in-lab usability study.

The rest of the paper is organized as follows. Section 2 briefly reviews the concept of business rules and a classification scheme of business rules, together with a motivating scenario that will be referred to throughout the paper to illustrate our approach. Section 3 is dedicated to the MoDAR development technique. Section 4 introduces the MoDAR development platform and the execution environment, along with the transformation between the models and the executable code. Section 5 evaluates our approach through detailed in-lab performance experiments and usability study. Finally, Section 6 surveys the state of the art and Section 7 concludes the paper.

2. Background

In this section, we review the concept of business rules and present a motivating scenario that will be used throughout the paper as an example.

---

1. In the rest of this paper, we call them business rules in general.
2.1. Business rules

According to the Business Rules Group\textsuperscript{3} “a business rule is a statement that defines and constraints some business. It is intended to assert business structure or to control or influence the behavior of the business”. From the IT perspective, a business rule is “an atomic piece of reusable business logic, that is specified declaratively”. Business rules make the knowledge about regulations, policies, and decisions explicit and traceable\textsuperscript{13}.

There are several classification schemes for business rules. For example, von Halle\textsuperscript{14} classified business rules into four groups: i) constraint rule that expresses an unconditional circumstance that must be true or false; ii) action enabler rule that checks conditions and initiates some actions upon finding the conditions true; iii) computation rule that checks a condition and when the result is true, provides an algorithm to calculate the value of a term, and iv) inference rule that tests conditions and establishes the truth of a new fact upon finding the conditions true.

Schacher and Grassle\textsuperscript{15} classified business rules into three categories: constraints, process rules, and derivation rules. Constraints and derivation rules have the same meaning as constraint rules and inference rules in von Halle’s classification scheme. Process rules ensure that some actions must or must not be executed in some situations. In this paper, we use von Halle’s classification scheme but extend the definition of action enabler rules so that they not only enable actions, but also disable actions. An action enabler rule is defined as a statement that checks conditions and initiates or disables some actions upon finding the conditions true.

It is worth noting that although computation could be taken as a kind of action, it is useful to separate them at the execution level: external actions need to be invoked to execute action enabler rules while computation rules can be executed internally by the rule engine.

2.2. A motivating scenario

Suppose that a travel agency StarTravel wants to provide a mobile application called SmartTravel to its customers. The main flow logic of this application is expressed in a BPMN (Business Process Modeling Notation)\textsuperscript{4} process as shown in Fig. 1. With SmartTravel, a customer can send travel requests to StarTravel via her mobile phone. Upon receiving a request, SmartTravel first validates the request, e.g., to see if the request contains valid departure and arrival cities, and if the validation fails, the customer will receive an SMS explaining why the request is failed. Otherwise, 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 can 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; if she is busy, an SMS will be sent instead.

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

\footnotesize{\textsuperscript{3} http://www.businessrulesgroup.org.  \textsuperscript{4} http://www.bpmn.org.}
The customer’s identity must be verified in the first place.

• Booking domestic flight and arranging international travel are merged to a single service because of company policy adjustment.

• The customer changes her preference and wants to rent a car only if the location of her accommodation is far from a famous tourist attraction.

• If there is a promotion, the cost needs to be re-calculated using the promotion rules.

• The customer wants to get the travel plan notification based on her presence: if she is busy (e.g., in a meeting), she wants the travel plan to be sent to her by SMS; otherwise she wants to get a direct phone call from the travel agency.

From the above scenario, we can see that a dynamic and changing environment and the user’s long tail of needs often require an application to be ready for change at any time. 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 minimum effort. If we follow the common SOC approach to implement SmartTravel as an executable BPEL process, the process structure needs 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 both services, and adjust the process structure. Another problem 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 rule, and if we change a business rule, the whole process needs to be changed.

3. The MoDAR approach

3.1. Overview

As illustrated in Fig. 2, the MoDAR approach consists of three main phases: i) the modeling phase, ii) the composition phase, and iii) the deployment phase. It is worth noting that these three phases match well with the three-layer abstraction of service-oriented systems proposed by Nitto et al. [16].
MoDAR PIMs, which include the base model, the variable model, and the weave model, are created in the modeling phase. Because adaptive systems are generally difficult to specify due to its high complexity and variability [17,18], we adopt the separation of concerns principle [19] to manage complexity and variability: a system model is divided into a base model and a variable model. The base model represents the relatively stable processing procedures, or flow logic, of the system; while the variable model represents the more volatile decision aspect of the business requirements.

We choose to use business rules to specify the variable model due to several reasons: i) a business process becomes more stable and reusable if we manage to abstract its complex branch structures into rules [13]; ii) business rules are easier to be changed than business processes because of their atomic and declarative nature [10,20]; iii) business rules are one of the core components in specifying requirements. Keeping business rules in the design model instead of translating them into complex conditional branches of a process not only prevents the paradigm shift from declarative rules to procedural processes, but also maintains the modularity and traceability of business rules.

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 and properties are used in constructing classes to build the necessary semantic connection between the two models. Any other class constructors such as intersectionOf, unionOf, and someValueFrom, which are mainly used in knowledge-based matchmaking and discovery, are not used in MoDAR. We also adopt an aspect-oriented approach to integrate the base model and the variable model using a weave model. This approach ensures the modularity of the base model and the variable model so that they can evolve independently [21]. MoDAR supports specifying cutting points on a process in which rules can be woven.

In the composition phase, the variable model is automatically transformed into Drools rules, and the weave model is automatically transformed into an abstract BPEL process. In this abstract process, at every join point, the invocation to a rule aspect is translated to a special Web service invocation. After the designer manually associates concrete Web services with abstract services in the process to implement their functionalities, the process is automatically transformed into an executable BPEL process. 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 we can freely add/remove/replace business rules defined in the modeling phase and then transform and redeploy them without terminating the execution of the process.

3.2. The MoDAR models

3.2.1. The base model

The base model is used to capture the flow logic, or procedures, defined in the requirements. To promote the notion of treating procedures and decisions as two independent elements in a requirement, we use a subset of a full-fledged workflow language as the language for specifying the base model, and in this subset, only procedural elements are included while excluding all the decision elements.
The structure of the base model language are shown in Fig. 3. To avoid creating yet “another” workflow language, we reuse the flow logic related elements defined in BPMN, while extending the original Business Activity element with properties for associating with the variable model.

In general, the base model language has two types of key elements: the Flow Object and the Connecting Object, where flow objects are processing entities and connecting objects specify the flow relations between flow objects. There are three types of flow objects: the Business Activity, the Event, and the Parallel Gateway. Business activities represent the main processing unit of a requirement. A business activity can be defined as variable to indicate that this activity is adaptive at runtime. Events have the normal meaning as defined in BPMN: they are happenings that affect the execution of a process (e.g., exceptions). To promote the separation between procedures and decisions, we only include parallel gateways—which represent the forking of process flow—in the base model while excluding all the other decision-related BPMN gateways. All the decision-related requirements will be captured by the MoDAR variable model. It is worth noting that the base model can still accommodate a full-fledged BPMN sub-process by encapsulating it in a composite activity on the condition that this composite activity is implemented as a single encapsulated component (such as a Web service), but the internal structure of this BPMN sub-process is not visible in the base model, and consequently any part of this sub-process is not supposed to be changed and is not changeable in the MoDAR runtime environment.

The detailed definition of the Business Activity is given as follows. A business activity is a tuple of name, isVariable, inputs, and outputs: \( t = \langle \text{name}, \text{isVariable}, \theta, \Omega \rangle \), where \( \text{name} \) is a finite set of names; \( \text{isVariable} \) indicates that the processing logic of this activity will later on be defined by the variable model and then changeable at runtime; \( \theta \) is a finite set of types, and every input or output of a business activity has a name and a type. The type of an input or output parameter must be a concept or datatype property in an ontology depending on whether this parameter is a complex structure or a primitive variable. For example, if the parameter is just the name of a customer, then we can use \( \text{Customer.name} \)—which is a string datatype property of the \( \text{Customer} \) concept—to annotate this parameter; otherwise if the parameter is a complex customer structure, we use the \( \text{Customer} \) concept to annotate it. The purpose that we associate an I/O parameter with an ontology concept or property 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 inter-operable; 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 flows. For example, the general steps of the SmartTravel process are illustrated in Fig. 4. We use dashed borderline to indicate variable activities.

Alternatively, we can re-engineer existing business processes and derive base models from them. A general guideline is described as follows. Firstly, because externally a base model can only have parallel gateways, a trunk of process that contains other kinds of gateways must either be encapsulated in a common composite activity, or be replaced by a variable activity. The difference is that the internal logic of a common composite activity is not changeable at runtime while that of the other is changeable. Clearly, if a trunk of process is identified as susceptible to change, it needs to be represented and replaced by a variable activity. It is worth noting that this trunk of process must be well-structured for it to be encapsulated in or replaced by an activity [22]. For example, a divergence gateway and its corresponding convergence gateway must be in the same trunk of process. Secondly, any activity in the original process—no matter atomic or composite—can be marked as variable for it to be changeable at runtime. But as we have mentioned above, the processing logic of this activity needs to be re-defined using the variable model.

Fig. 5 gives an example of a derived base model from the SmartTravel BPMN process. There are four trunks of processes that are identified as variable activities \( V \), and the Calculate Price activity is also marked as variable to enable runtime change/adaptation. It is worth noting that in the derived base model, the logic of both the exception throwing and handling has been included in \( V \). As such, the base model also helps in isolating the exception handling logic from the main business logic.

3.2.2. The variable model

The variable model is used to capture the decision aspect of a business requirement, which is changeable at runtime. A variable model consists of a set of business rules where each rule \( r \) is defined by a 3-tuple: \( r = \langle \text{type}, \text{condition}, \text{action} \rangle \). The type property specifies which category a rule belongs to basing on the rule classification scheme discussed in Section 2.1. A rule type can be either constraint, or computation, or inference, or action. Our rule definition follows the typical
Fig. 5. Deriving base model from BPMN process.

event-condition-action (ECA) pattern but the event part is specified in the weave model (see the next subsection for details) because the triggering of rules is determined by point cuts in the aspect.

Following the propositional logic based constraint languages such as WSCoL [7] and JML [23], we have designed a high-level rule language and its associated graphical editing tool (see Section 4.1.1) to facilitate the specification of business rules. The syntax of this rule language is defined as follows:

\[
\begin{align*}
\text{<rule>} & ::= \text{<type>}, \text{<cond>}, \text{<action>}
\text{<cond>} & ::= \text{not <cond>} | \text{<cond> and <cond>} | \text{<cond> or <cond>} | \text{<term> <relop> <term>}
\text{<term>} & ::= \text{<property> | <term> <arop> <term>} | \text{<const> | <fun> (<term> <term>*)}
\text{<property>} & ::= \text{<concept>_<n>?.<obj_prop>*.<datatype_prop>}
\text{<relop>} & ::= \text{less than | less than or equal to | equal to | greater than or equal to | greater than}
\text{<arop>} & ::= + | - | * | /
\text{<n>} & ::= 1 | 2 | 3 | ...\\
\text{<fun>} & ::= \text{<pref> | <usrdef>}
\text{<pref>} & ::= \text{abs | replace | substring | sum | avg}
\text{<action>} & ::= \text{(<act>)} | \text{<property> | <concept>_<n>?|<term> | <act>} | \text{<activity> | <activity>} | \text{Abort | <activity> Then Abort}
\end{align*}
\]

There are two main features in this rule language: i) ontology concepts and properties are introduced in the specification of both the condition and the action of a rule; ii) user defined functions, such as the distance function for calculating the distance between two places, are allowed in defining the condition of a rule. The benefit of the second feature is straightforward—a domain specific function library can be built to facilitate the definition of complex rules. Next we focus on the discussions of the first feature of our rule language.

Because of the atomic feature of rules, in many situations, only one instance (or variable) of the same concept/type is involved in the definition of a rule. In such cases, the name of an ontology concept is directly used to represent one of its instances to bring certain convenience to the rule author. For example, to define the condition “if the customer is a frequent...”
flyer", we can just write the following natural-language-like condition expression: "Customer.isFrequentFlyer equal to true", in which the ontology concept Customer actually means a specific customer in the context of the rule. If more than one instance of the same concept is needed in a rule expression, number subscriptions, such as Customer_1, Customer_2, are used to identify a specific instance.

Based on Web Ontology Language (OWL) [24], an ontology concept could be a complex structure having both object properties and datatype properties, where an object property navigates to another concept in the ontology and a datatype property has a specific primitive data type such as integer, boolean, or string. For example, suppose the Customer concept has an object property contact whose range is the concept Contact, and phoneNumber is a string datatype property of Contact. Only datatype properties are allowed in defining condition terms because operations on objects are not defined in the context. For the action part, we can either assign the result of a term expression to a variable, or assign the result of the invocation of a business activity to a variable. We also predefine four special actions to express the semantics of disabling an activity and abort the execution of the process:

- Skip <Activity>: skip an activity;
- Skip <Activity> Then <Activity>: skip an activity then do another activity;
- Abort: abort the running of the process;
- <Activity> Then Abort: invoke an activity before aborting the running of the process.

The following are two rule examples (more examples are demonstrated in Fig. 6):

**R$_1$**: If the departure airport or arrival airport of a travel request is empty, stop processing the request:

[Cond] DepartureAirport.name equal to "" or ArrivalAirport.name equal to ""
[Action] sendSMS(Customer.phoneNumber, "departure or arrival airport cannot be empty.") Then Abort.

**R$_2$**: If a customer is a member of StarTravel, give him a 10% discount:

[Cond] isMember(Customer.id)
[Action] Booking.price = Booking.price * 0.90.

The first example is a constraint rule. If the name of the departure airport or the arrival airport is not set, then the condition is evaluated to true and the action will be invoked. For the action part, first the sendSMS business activity is invoked and then the execution is aborted.

The second example is a computation rule. In its condition, a function isMember is used; if the result is true, the price property of Booking, which represents a variable in the context of the rule, is assigned a new discounted value.
3.2.3. The weave model
The decision logic defined in the variable model is associated to the base model by the weave model, which is composed of a set of aspects. Each aspect α weaves a rule set to a business activity in the base model: α ∈ \{Before, Around, After\} × T × 2^R, where T is the set of business activities defined in Section 3.2 and 2^R is the set of rule sets.

Similar to AspectJ, we also identify three types of aspect: before aspects, around aspects, and after aspects. An aspect is always associated with a business activity. Both before aspects and around aspects are activated before the execution of the associated activity, but if an activity has an around aspect, this activity will not be executed after the around aspect. From the perspective of the ECA pattern, \(event ∈ \{Before, Around, After\} × T\) 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 pre-defined 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 (e.g., \(R_1\) in previous section) can use these two properties in defining itself.

Fig. 6 is an example of the weave model that contains six aspects: one before aspect, four around aspect, and one after aspect. This example is able to accommodate all the changes arose from the requirements discussed in Section 2.2. For instance, to include the additional action of checking the customer’s identity, a before aspect can be weaved to the Check Request variable activity. For the Book Flight variable activities, if StarTravel decides to merge its domestic and international departments, we can just update the rules in its around aspect to reflect this change, and the after aspect weaved to the Calculate Price variable activity can be used to facilitate changes in the pricing policy.

4. The MoDAR platform

In this section we introduce the details of the MoDAR platform and also discuss how to use it to define MoDAR models (i.e., the base model, the variable model, and the weave model), and how these models are transformed to executable code.

4.1. The development environment

The MoDAR development environment has two main components: i) the Process Modeler for graphically modeling the base model, the rules, and the weave model, and ii) the Association and Transformation Tool for associating Web services with activities defined in the base model and for generating and deploying executable code. To facilitate process and rule definition, the development environment also has a Business Domain Explorer component for graphical exploration of domain ontologies defined in OWL (the OWL files are created using ontology editing tools such as Protégé).

4.1.1. The Process Modeler

The Process Modeler offers a visual interface for defining the base model, the variable model, and also the weave model. As shown in Fig. 7, the main canvas in the middle of the graphical interface of the Modeler is used to define and display the base model. Essential BPMN constructs (including Activity, Parallel Gateway, Start Event, End Event, and Sequence Flow) and also Variable Activity (the dashed box) are displayed as a list of buttons on top of the canvas for visually creating the base model. By clicking a specific activity in the base model, we can define its I/O semantics in the bottom pane. If we select a concept or datatype property from the drop-down menu that contains all the concepts and datatype properties 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 ArrivalAirport, and one output parameter with type FlightBooking.

The middle canvas is also used to define the weave model. As shown in Fig. 7, the left pane of the process modeler contains a list of business rules that can be drag-and-dropped to an activity and becomes its before/around/after rule. It is worth noting that if a rule is associated with an activity, this activity automatically becomes a variable activity, and all the parameter variables of this activity are also visible to the rule.

Fig. 8 is the XML code for the CorrectAirports before aspect. This aspect is defined inside the Book Flight activity as it is associated with this activity. An action enabler rule CorrectAirports is defined inside this before aspect with the Location attribute indicating the location of the rule file. Also, an activity that will be invoked by the rule is also defined inside the rule element. This activity ContextService has been associated with a concrete service SMSCustomer.

A new business rule can be created in the “Business Rule Editor” dialog box, which will appear if we right-click one of the folder icons in the left rule repository pane and select “New Rule” from the pop-up menu. The rule editor uses the concepts in the domain ontology to define the condition and action components of a rule. It is worth noting that all the I/O parameter variables in the base model that are visible to a rule will be automatically bound to the corresponding concepts or properties in the rule, and if a rule needs to use variables other than the parameters of the associated activity, we can use the variable visibility tab in the bottom pane to make additional variables visible to this rule.
4.1.2. 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 activities defined in both the base model (the BPEL process) and the variable model (within the action part of rules). Second, it is used to automatically generate executable code and deployment scripts from the models defined in the modeling phase. In the current implementation BPEL was selected as the targeted executable process language and Drools as the targeted executable rule language.

Fig. 9 shows the graphical interface of the Association and Transformation Tool. The left pane of the graphical interface is a directory of available Web services. A user can drag-and-drop a Web service onto a process activity or an aspect to associate the Web service with this activity or a rule activity in the aspect. Since semantic service discovery and match is not the focus of our work, currently only a basic manual association method is supported: a Web service can be associated with an activity if and only if its IO parameters exactly match the IO parameters of the activity in terms of types and number. As shown in Fig. 9, all the process activities and aspects with services correctly associated are marked with a “tick” (if the action of a rule does not need external services, it is also taken as correctly associated), while a question mark is shown if an activity (either in the process or in a rule) is not correctly associated with a service.

http://protege.stanford.edu/.
Fig. 10 shows the generated Drools rule code for rule $R_1$ discussed in Section 3.2.2. In order to keep the invocation of Web services associated with activities defined within rules self-contained, service information for Web services associated with activities defined within rules are encoded directly into the rule code. First, the bindings for ontology classes used in the rule as well as an enabler helper-class are defined (Lines 5–8), followed by the condition statement as translated into Drools syntax (Lines 10–11). As rule $R_1$ specifies an Abort action, if the condition statement is satisfied the global abort status of the aspect is then set to true using the enabler helper class (Line 14), followed by the Web service associated with the SendSMS activity being invoked, again using the enabler helper class (Lines 16–32).

On transformation of the weaved process model, the base model is transformed into a BPEL process, with BPMN elements such as Start, Event, and Activity translated directly into their corresponding BPEL constructs (i.e., receive and invoke). A more detailed and comprehensive solution to translating BPMN to BPEL can be found in [25]. Variable and Activities however undergo a slightly more involved transformation process: first, the Before and Around aspects associated with the same variable activity are merged into one aspect service, and a merged Drools DRL file is created in the rule repository for this aspect service. A merged DRL file is also created for every After aspect. BPEL code that invokes the service corresponding to the variable activity is then generated, preceded and/or followed by invocations to the associated aspect service. It is worth noting that because an aspect may contain multiple rules, when more than one rules are activated at the same time, some conflict resolution strategy can be configured in Drools, for example Salience, or FIFO.\(^6\)

There is an “Advanced Adaptability” transformation option that can bring more adaptability to our approach, but it also comes with some performance cost. A user can ask the transformation tool to always generate two aspect service invocations before and after a variable activity, no matter a corresponding aspect service is defined or not. With this option on, we intentionally reserve two extension points for every variable activity so that additional business rules can always be dynamically added to these two extension points even if an aspect is not defined at process design time.

The aspect service takes as input a URI to the merged Drools DRL file, along with the values and ontology class names of all variables involved in the aspect. It returns two Boolean values corresponding to abort and skip evaluation outcomes, as well as the values of all variables that may have been updated based on rule evaluation. Three IF conditions are finally inserted around the variable activity invocation to handle abort and skip actions based on the return of the aspect service.

Fig. 11 illustrates the above described transformation, showing a BPEL code snippet generated from the weave model that corresponds to the “Book Flight” variable activity. Here, the aspect service is invoked on Lines 1–4 for Before and Around

\(^6\) [http://legacy.drools.codehaus.org/Conflict+Resolution](http://legacy.drools.codehaus.org/Conflict+Resolution)
rule "Valid Airports"
dialect "java"

when
$enabler : Enabler()
$Customer : Customer()
$DepartureAirport : DepartureAirport()
$DestinationAirport : DestinationAirport()

DestinationAirport(name != "") AND
DepartureAirport(name != "")

then
$enabler.setAbort(true);

try {
    String[] wsInfo = { "http://localhost:8080/
        contactservice",
    "smsCustomer", "ContactService",
    "ContactServicePort",
    "http://localhost:8080/
    ContactService/ContactService?wsdl"};

    String[][] varInfo = {{ $Customer.getPhoneNumber()},
        "departure or arrival airport cannot be empty."}];
    String[][] varNames = {{ "CustomerNumber" },
        { "ReturnMessage" } ];

    $enabler.runService(wsInfo, varInfo, varNames);
} catch (Exception e) { e.printStackTrace() };

end

Fig. 10. Drools rule code corresponding to rule $R_1$.

<assign> ...service input... </assign>
<invoke partnerLink="rule" operation="RuleEngineWS"
    outputVariable="ruleresponse" ... />
<assign> ...service return... </assign>

<if>
    <condition>contains($ruleresponse.ruleOut/rules[1],
        "true")</condition>
    <sequence><throw name="cancelProcess" faultName="cancelation"/></sequence>
</if>

<if>
    <condition>contains($ruleresponse.ruleOut/rules[2],
        "false")</condition>
    <sequence>
    <assign> ...service input... </assign>
    <invoke partnerLink="flightservice"
        operation="FlightBooking" ... />
    <assign> ...service return... </assign>
    </sequence>
</if>

<if>
    <condition>contains($ruleresponse.ruleOut/rules[1],
        "true")</condition>
    <sequence><throw name="cancelProcess" faultName="cancelation"/></sequence>
</if>

<assign> ...service input... </assign>
<invoke partnerLink="rule" operation="RuleEngineWS"
    outputVariable="ruleresponse" ... />
<assign> ...service return... </assign>

<if>
    <condition>contains($ruleresponse.ruleOut/rules[1],
        "true")</condition>
    <sequence><throw name="cancelProcess" faultName="cancelation"/></sequence>
</if>

Fig. 11. SmartTravel BPEL code snippet.
aspects and Lines 24–27 for the After aspect (suppose we use the “Advanced Adaptability” option). The variable activity invocation (Lines 17–20) is wrapped in an IF conditional block (Lines 13–22) that allows it to be skipped based on the outcome of the first aspect service invocation, with IF conditional blocks on Lines 6–11 and Lines 29–34 present to allow the entire process to be aborted via throwing a cancelProcess exception based on the outcome of the Before/Around and After aspects respectively. Finally, to facilitate the deployment of the transformed weaved process, ant\textsuperscript{7} scripts are generated that automate the deployment of the BPEL and Drools rules code to the corresponding engines.

4.2. The runtime environment

Fig. 12 is an anatomy of the MoDAR runtime environment. The bottom level of the anatomy includes the main components of the runtime environment: a Drools engine (Drools 5.0), a BPEL engine (Riftsaw 2.2.0), and a generic aspect service that encapsulates the rule invocation logic. The aspect service is written in Java and exposed as a Web service for the BPEL process to invoke. Every time when an aspect in the process is reached for execution, the aspect service is invoked and corresponding variables (including the IO parameters of its associated activity and user selected variables) are passed from the process to it; these variables are used in the execution of the rules of the aspect. After all the rules in the aspect are executed, these variables are updated and passed back to the process. As discussed in the above subsection, a Drool rule can invoke external Web services to implement action-enabler business rules. In case an exception is raised when executing a Drool rule or invoking an external Web service, the exception will be propagated by the aspect service to the base process for exception handling. For error message traceability between the Drools engine and the BPEL engine, our current implementation relies on the built-in logging functionality of Drools.\textsuperscript{8} A further extension to the aspect service would be passing the error message or error code generated by the Drools engine directly to the BPEL engine.

The anatomy of the MoDAR runtime environment reflects the dynamic adaptability, i.e., what can be changed and what cannot be changed at runtime, of our approach. As we can see, at runtime both the condition and action part of a rule can be changed, and rules can be added and removed from an aspect. But such a change is limited to the variable set exchanged between the process and an aspect, which means if we modify or add a rule in an aspect that requires additional variables than the preselected variables, such change will not take effect. But this restriction can be removed if the process designer chooses to transfer all the variables defined in the process to the rule engine. The designer may use this option in some situation if he/she thinks that the performance cost of exchanging additional variables is acceptable (the performance cost w.r.t. the number of exchanged variables can be found in Section 5.1).

At runtime, clearly we can not introduce new aspects to non-variable activities as aspect services are not defined for non-variable tasks in the generated BPEL process code. As for variable activities, if the “Advanced Adaptability” option is on, we can always introduce new aspects dynamically; otherwise an around aspect can be added to a variable activity having a before aspect predefined (vice versa), because a before aspect and an around aspect are jointly transformed to one aspect service.

5. Evaluation

5.1. Performance study

Our approach brings the capability of dynamic adaptation to BPEL processes and clearly also brings some performance overhead. Based on the anatomy of the MoDAR runtime environment illustrated in Fig. 12, we can see that the performance

\textsuperscript{7} http://ant.apache.org/

\textsuperscript{8} http://docs.jboss.org/jbpm/v5.2/javadocs/org/drools/jbpm/KnowledgeRuntime-LoggerFactory.html
overhead mainly comes from two places. Firstly, the aspect service is an extra Web service that needs to be invoked for each aspect, and there are variables exchanged between the process and the aspect service. Secondly, every rule is actually the externalization of a conditional statement in the process, and executing a rule may be slower than executing the equivalent conditional statement.

Several experiments have been conducted to evaluate the actual performance impact of the approach. The MoDAR runtime environment was run on a PC with an Intel Core i7 860 2.80 GHz CPU and 4 GB of RAM. Riftsaw 2.2.0\(^{10}\) running on top of JBoss-Application-Server-5.1.0GA was used as the BPEL engine, with Drools-5.0\(^{11}\) used as the rule engine.

In the first experiment, we tested the impact of invoking a single empty aspect service with various numbers of randomly generated primitive type variables passed. Every setting was tested five times and the average execution time of an empty aspect service w.r.t. the number of passed variables is shown in Fig. 13. As we can see, it costs 22 ms to invoke an empty aspect service without passing any variables and costs 32 ms to invoke an empty aspect service with 100 variables passed to it. This result shows that the variable exchange between the Riftsaw BPEL server and the Drools server is quite fast, and there is only 10 ms increase from passing no variable to passing 100 variables. The reason could be that these two servers are two components that both run inside the same JBoss application server.

In the second experiment, we compared the performance of an aspect and its equivalent BPEL conditional statements with various number of rules at various conditional expression complexity level. We fixed the number of exchanged variables to 100, with each variable being assigned a random primitive type and value. According to the result of the first experiment, passing 100 variables brings about 10 ms overhead to the overall aspect service invocation time. Because the action part of a rule (e.g., Web service invocation) takes the same time to run no matter it is invoked from a rule engine or from a process engine, we used empty action for all the rules and conditional statements. In addition, we took the number of variables evaluated in a conditional expression as the complexity level. For example, at complexity level 10, 10 variables are randomly selected from the 100 candidates and the value of each variable is tested both in the condition part of a rule and in the corresponding BPEL <if> statement. In the experiment, we selected 1, 10, and 30 as the representative complexity levels, and every setting was tested 5 times.

The result of this experiment is shown in Fig. 14. As we can see, replacing an aspect with its equivalent IF statements always brings some performance overhead. At complexity level 1, the average overhead is 97 ms for 10 rules (i.e., 109 ms for invoking the aspect service minus 12 ms for executing the IF statements), and 345 ms for 100 rules. At complexity level 30, the overhead is 124 ms for 10 rules and 408 ms for 100 rules. From this experiment we can see that the overhead of invoking an aspect service varies from 97 ms for a simple case to about 408 ms for a very complex case.

In order to estimate how much overhead a weaved MoDAR process may bring in real-life, we also compared the execution time of two implementations of the running scenario: one is a manually composed BPEL process and the other is the weaved process (as shown in Fig. 6) 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 18,265 ms to execute and the weaved process (with six aspect services) takes about 18,971 ms. We can see that the overhead of the weaved process compared to the pure BPEL process is 706 ms (or 3.87%), and the average overhead per aspect service is 118 ms in this case. If we turn the “Advanced Adaptability” transformation option on, the weaved process would have 1072 ms (or 5.87%) overhead.

---

9 If an activity has both a before aspect and an around aspect, they will be combined to one aspect Web service invocation at runtime.
10 http://www.jboss.org/riftsaw/.
11 http://www.jboss.org/drools/.
5.2. Usability study

To validate the applicability of the MoDAR approach, we implemented it as a case study in a medium-size medical clinic with 70 full-time staff members. The interview after running the MoDAR processes for three months indicates the efficiency of our approach in handling rapidly evolving policies and rules that govern clinics and hospitals within the medical industry.

To further study its usability, we conducted an in-lab controlled experiment, testing the efficiency of users in developing and adapting a service process using both MoDAR and a standard BPEL graphical designer. Participants in the study were five Information Technology research students who have studied BPEL in a Web-related subject and have some experience in developing Web services using BPEL as we assume that the typical users of MoDAR are software engineers who have knowledge in developing BPEL processes. It is worth noting that using small number of participants to do usability study has been justified by Nielsen [26] and has been successfully applied in [27,28] for example.
Table 1
Results of the user efficiency study (unit: Minute).

<table>
<thead>
<tr>
<th>Participant No.</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MoDAR</td>
<td>BPEL</td>
<td>MoDAR</td>
</tr>
<tr>
<td>1</td>
<td>8.17</td>
<td>13.67</td>
<td>0.67</td>
</tr>
<tr>
<td>2</td>
<td>8.92</td>
<td>9.83</td>
<td>2.67</td>
</tr>
<tr>
<td>3</td>
<td>12.08</td>
<td>24</td>
<td>1.08</td>
</tr>
<tr>
<td>4</td>
<td>14.38</td>
<td>6</td>
<td>2.25</td>
</tr>
<tr>
<td>5</td>
<td>13.75</td>
<td>3.25</td>
<td>2.33</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>11.50</td>
<td>11.35</td>
<td>1.80</td>
</tr>
</tbody>
</table>

The participants have been asked to do the following typical tasks in both MoDAR and Eclipse BPEL Designer: 12

- Task 1 (development): develop the “New Medical Image Request” business process (which is used in the case study).
- Task 2 (adaptation-change rules): change the process to adapt to the new requirement: “If examination type is X-ray, no doctor is needed”.
- Task 3 (adaptation-add new rules): change the process to adapt to the new requirement: “Assign English speaking Technician for non-Japanese patients”.

The time each participant used to complete each task is presented in Table 1, and Fig. 15 is the box plot based on the data in Table 1 to show the spread of the data. For Task 1, there is no apparent user performance difference between using MoDAR and BPEL Designer. One of the participants even used less than 5 minutes to complete the task in BPEL Designer, although we have a more compact spread for MoDAR. For Task 2 and Task 3, on average MoDAR is better performed than BPEL Designer. When excluding the outliers, MoDAR is still slightly better performed than BPEL Designer. From this experiment, we can see that all the participants can use both MoDAR and BPEL Designer to complete the development and modification of a business process in a similar time frame.

To collect user’s perception of ease of use, ease of learning, and satisfaction of MoDAR, participants in our study completed the following short survey. We indicate the rationale for each question in brackets. The users were not shown the rationale for the questions. The survey contains the following seven questions:

- Q1: During the development process, did you know what you could do at each step of the process? (ease of use of the MoDAR MDD approach)
- Q2: Was the MoDAR platform easy to use? (ease of use of the MoDAR graphical user interface)
- Q3: Did the MoDAR platform help you be more productive in developing business processes? (usefulness of the MoDAR development platform)
- Q4: Did the MoDAR platform help you be more productive in making changes to the existing processes? (usefulness of the MoDAR dynamic adaptation feature)

12 http://www.eclipse.org/bpel/.
The survey has multiple-choice Likert scale responses ranging over values such as strongly disagree, disagree, neither, agree, and strongly agree. To produce a numerical score, each survey response is mapped into the range of 1 to 5, with five representing the greatest degree of satisfaction. All of the responses are summed up to produce a score ranging from 5 to 25 for each question.

All the five participants completed the survey and the results are shown in Table 2. The results indicates that in general all the participants were satisfied with MoDAR, with no score less than 3, and no total score for each question less than 19 (76%). Among all the questions, Q5 (Ease of learning) received the highest score (24 out of 25). Q3 (Usefulness) was relatively the least scoring one, which is consistent with the usability experiment in terms of process development efficiency.

6. Related work

Dynamic and adaptive processes are one of the key research challenges in service-oriented computing [2]. In general, existing approaches on dynamically adaptive service processes can be classified into two main categories. The first category of work focuses on ensuring the non-functional quality attributes of the service process, for example, automatically replacing a low-performing component service with another better performing one or recovering the execution of the process from a service failure. The other category of work, where our work falls in, aims at coping with the functional changes raised from both business requirements and environmental context, and bringing dynamic adaptability (or agility) to service processes.

In Table 3, we compare dynamic adaptation approaches for service processes based on a set of carefully selected classification metrics. We choose these approaches as most related work because all these approaches use BPEL—the mainstream service process execution language—as the base technology and enhance it with various degree of dynamic adaptivity.

According to [29], adaptation can be static or dynamic, manual or automatic, and proactive or reactive. We only consider dynamic adaptation approaches. Manual adaptation means human efforts (e.g., writing adaptation code) are needed...
to instruct the system how to do adaptation, while automatic adaptation can be performed by the system according to predefined adaptation conditions (e.g., replacing a service to an alternative one if it is not available). Usually functional adaptation approaches have to be manual because human efforts are indispensable to translate new requirements to system functions, and non-function adaptation can be automatic because the semantics of adaptation actions, such as comparing and replacing, are usually predefined. Proactive or reactive adaptation means the adaptation is happened before or after a specific event. We do not use this metric because we take all the approaches as reactive in terms of adaptation as a reaction to change, although such distinction could be used to indicate recovery approaches, which are not the focus of our comparison.

McKinley et al. [30] pointed out that three aspects must be considered in project related to adaptability: how, where and when the adaptation takes place. We summarize the adaptation mechanism (“how”) into three categories: i) message interception, in which adaptation takes effect by intercepting BPEL messages; ii) late binding, in which a proxy service is responsible for binding the real service at runtime; and iii) aspect injection, in which aspects are weaved into the BPEL code to do adaptation at runtime. The point “where” the adaptation code is inserted can be in the BPEL process source code, or in the BPEL engine source code, or in both places. Approaches that only change the BPEL engine have no impact to BPEL processes but their capability is limited to replacing component services. All the approaches in Table 3 apply adaptation at runtime (“when”) so we do not include this aspect in the table. Moreover, some approaches provide programming level support while some others also provide high-level design tools and languages. Because rules play an important role in our approach, we also compare if a rule engine is used in a particular approach. Finally, we indicate which approaches have dynamic extensibility [31]—the ability to change adaptation strategies to cater for new requirements at runtime, which is a highly desirable feature for adaptation approaches.

Among the approaches that focus on functional adaptation, Charfi et al. [20] use AO4BPEL [32,8]—an aspect-oriented extension to BPEL that supports dynamic weaving of aspects in BPEL processes—to weave business rules to a BPEL process. The main differences between this work and MoDAR are that although they discussed the benefits of using a rule engine to manage and execute rules, a rule engine is not integrated in their approach and business rules are manually mapped to BPEL conditional. Their work focuses on the execution language and environment while MoDAR is also a systematic engineering approach with a visualized design tool-set. In their approach, each rule is mapped to one aspect while several rules can be included in one aspect in MoDAR.

Rosenberg and Dustdar [33] proposed a runtime environment where both a BPEL engine and a rule engine are connected to a service bus. Dynamic adaptation is achieved through intercepting the messages exchanged between the process and a partner service and invoking business rules running on the rule engine before and after the execution of the partner service. Compared with MoDAR, there are some limitations in their approach. It is not possible to disable or replace a partner service at runtime because rules are inserted before and after the invocation of a partner services. In addition, rules can only be applied on the data in intercepted messages. Paschke and Teymourian [34] discussed a rule based business process execution environment where a rule engine is deployed on an ESB (Enterprise Service Bus) and exposed as Web services. Dynamic adaptation is achieved by explicitly defining and integrating Rule Activities, which invoke the rule service, in the BPEL process, and rules can be modified and applied without redeploying the process. Although they share the same idea with us as to introducing rules to business processes in order to bring agility and adaptability, their approach of explicitly defining rule service in the BPEL process violates the separation of concerns design principle and could possibly increase the complexity in defining and maintaining such paradigm-mixed processes, which itself may need to be changed from time to time.

Karastoyanova and Leymann [35] also proposed to use aspects to enhance the dynamic adaptability of BPEL processes. In their approach, the BPEL engine is extended with an event publishing framework so that lifecycle events can be propagated to external components and the aspects representing the adaptive logic are mapped to subscriptions to appropriate process lifecycle events. This work mainly focuses on the execution environment. To use this approach, extensions to BPEL engine is necessary. In comparison, the MoDAR approach also includes a systematic engineering approach to developing adaptive BPEL.

Among the non-functional adaptation focused approaches, many of them [36–41] intended to provide a late (dynamic) binding solution in order to select the most suitable (e.g., having the best quality attributes) service, or to replace a misbehaved (e.g., invocation fail or running overtime) among a set of functionally equivalent services. Proxy services are frequently used to implement late binding. They are embedded in the original BPEL process to do runtime discovery and binding, either statically or dynamically configurable.

Aspect-oriented paradigm is also frequently used in non-functional adaptation. Compared with dynamic binding approaches, aspect-oriented approaches have the potential to introduce richer adaptation logic to the process than just selecting or replacing component services. The main difference between aspect-oriented functional and non-functional adaptation approaches is that non-functional approaches usually need to provide monitoring mechanisms to sense the quality attributes of component services, and aspect logic is usually predefined (although it could be changed at runtime) to provide automatic adaptation capabilities. Charfi et al. [31] proposed a self-adaptation plug-in service composition architecture based on AO4BPEL. Moser et al. [4] proposed a execution environment called VieDAME that has quality attributes monitoring and dynamic adaptation functions. It is worth noting that VieDAME is also a message-intercepting approach so the adaptation is limited to interactions with partners. Baresi and Guinea [7] proposed an aspect-oriented supervision framework for BPEL with dedicated monitoring and recovery languages for defining aspects.
In the general research area of software architecture, model-driven development is combined with aspect-oriented methodology to manage the vast number of possible runtime system configurations of dynamic adaptive systems. In [18], 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 [42], the authors propose an aspect-oriented approach for implementing self-adaptive system on the Fractal platform [43]. 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 [43] and OSGi [44], our targeting platform is BPEL, the de facto industry standard language for service composition.

Recently, Dynamic Software Product Line—DSPL, first introduced by Hallsteinsen et al. [45] as a new trend to develop dynamically adaptive software products—has attracted much attention in the software engineering community. Dinkelaker et al. [46] propose the concept of dynamic features, which can be linked to dynamic aspects to realize runtime system variability. Parra [47] propose to divide the products in several modules called aspect models and then dynamically adapt a product using runtime weaving. Potentially, it is possible to enhance MoDAR with the feature modeling option as a mechanism to manage the versions of a dynamic process and its associated variable models.

7. Conclusion

In this paper, we have presented MoDAR, a model-driven approach for developing dynamically adaptive BPEL-based service-oriented systems using aspects and rules. We have introduced the methodology of separating the variable part from the stable part of a process and modeling the variable part as business rules, as well as weaving these rules into the stable base process. We have also developed a platform using a number of the state-of-the-art technologies that support the model-driven development and execution of rule-weaved BPEL processes. The targeting system is dynamically adaptable in the sense that rules as the variable part of a system can change at runtime without affecting the base process. Experimental results have shown that systems developed and generated with MoDAR have slight performance overhead but significantly simplifying the development process and achieving runtime adaptability.

Our future work will focus on tackling several limitations identified during the real-life medical clinic case study. In particular, we will enhance the re-usability of rules in the development tool, and investigate runtime validation techniques. We plan to develop new techniques on top of the MoDAR platform to facilitate the development of context-aware applications. Furthermore, we also plan to translate MoDAR to a formal model to improve the expressiveness of the MoDAR models, and to support the property verification of MoDAR applications.

References


