

Guiding the Service Composition Process with Temporal Business Rules

Jun Han¹, Yan Jin¹, Zheng Li¹, Tan Phan¹ and Jian Yu²

¹Faculty of ICT, Swinburne University of Technology, Hawthorn, VIC 3122, Australia
{jhan, yjin, zli, tphan}@ict.swin.edu.au

²Department of Automation and Information, Politecnico di Torino, Turin, 10129, Italy
jian.yu@polito.it

Abstract

Service composition has become an important paradigm for building distributed applications and e-business processes. While effort has been reported to verify a posteriori whether a given composition such as a BPEL schema satisfies the predefined behavioural properties, little effort has been made to utilise the properties to assist the designer in developing a correct service composition in the first place. This paper reports our first attempt towards this goal by presenting a framework and associated techniques to provide automated guidance to the designer during the composition design process. The guidance can be suggestions on the next valid steps in the business process, identifications of missing/misplaced steps, and/or propositions for inserting, deleting or re-ordering activities. The guidance is provided based on the temporal business rules which state the temporal/sequential relationships between business activities.

1. Introduction

Service oriented computing presents a promising paradigm for building distributed software applications across organisational boundaries. It promotes the use of self-describing and platform-independent services as the fundamental computational elements to compose cross-organisational business processes. Typically, a service composition is done in the context of Web services and is specified in a composition language such as BPEL [4] or BPML [2].

In the process of application development, it is essential to ensure that the service composition being developed possesses the desired behavioural properties. These properties conjointly reflect the standards and well-accepted practice in a business domain as well as the business rules and guidelines set by each participating organisation. A service composition that fails to conform to these properties will either violate the generally agreed rules at business domain level or contain tasks that a service provider can not carry out.

Various research efforts have been reported to tackle

this problem. Most of them focus on post-composition verification, which checks the conformance of a completed service composition to the predefined properties. Therefore, these approaches provide little help during the service composition process. There also has been much research effort in automatic generation of service compositions. At current stage, however, the composition design is predominately a human-driven process and is mainly accomplished by system developers. It is a complicated and tedious task to manually design a service composition that has to comply with a set of complex properties. Rather than repeating the design-verify-redesign loop, it is better to have a tool that can automatically and proactively provide guidance and decision making support during the design process, so that service composition designers could efficiently compose a valid application in the first place. Providing continuing guidance through a composition design process would bring the following benefits: Firstly, it frees designers from the task of frequent conformance checking between the current design and predefined properties during the process, so that designers can focus on the implementation of business logic. Secondly, designers can have immediate feedback or hints on how to progress in the next step, which ensures validity of the current design. Thirdly, the mistakes or design errors can be identified on-the-fly during the composition process, much earlier than existing post-composition analysis approaches. Fourthly and most importantly, when a mistake/error is identified, the designer will be informed of the reason and with suggestions on how to address it.

In this paper, we propose a framework and associated techniques to automatically provide guidance to BPEL designers during the service composition process. The guidance is provided based on the *Temporal Business Rules*, which prescribe the occurrence or sequence pattern of business activities in a business domain. For each composition step, the tool automatically checks the conformance of the BPEL design/schema to the given rules, using the Finite State Automata (FSA) based checking approach developed in [17]. Then it analyses the checking results to determine what guidance to

provide towards a correct and more complete service composition. In general, the guidance can be suggestions on the next valid steps in a business process, identifications of missing steps or misplaced activities, and activity re-ordering. This paper presents the various strategies and algorithms we developed to automatically generate such guidance during the service composition design process. Our work is aimed at ease adoption by practitioners, who usually do not have the in-depth training on intricate temporal logics or formal languages. The proposed framework thus features an *intuitive* pattern-based approach to specifying temporal business rules and an *automatic* approach to checking BPEL schemas, which are used to generate guidance.

The rest of this paper is structured as follows. Section 2 presents an overview on the framework for guidance provision. Section 3 elaborates the strategies and the associate algorithms that automatically generate proactive and remedy guidance during the BPEL design process. The implementation of the framework is introduced in section 4. Section 5 discusses the related work and we conclude the paper in Section 6.

2. A Framework for Guidance Provision

Before discussing the strategies and algorithms for guidance generation in detail in section 3, we present an overview on the key components in the framework to provide a preliminary context.

Our framework is built on the following philosophy: in-progress model checking of a service composition can provide on-the-fly decision making support to designers via proactive guidance on the next valid steps, and real-time detection/correction of invalid/missing steps. More specifically, the framework is able to list the next valid steps based on a current selected point in a BPEL schema. Additionally, at each step of a service composition, the framework can check whether a syntactically correct BPEL schema conforms to all given temporal business rules and generate remedy suggestions if any nonconformity is detected. Figure 1 depicts overall structure of the framework, where shadowed ovals signify three key components: WSDL Annotation, Rule Specification and Guidance Generation.

WSDL Annotation. A service composition process normally starts with a set of business activities, which may be defined by domain experts or drawn from domain and process ontologies to ensure a consistent interpretation [9]. In most cases, the business activities are implemented in Web services and are bound with operations defined in the corresponding WSDL files. We establish a mapping between business activities and Web service operations by annotating the WSDL file, so

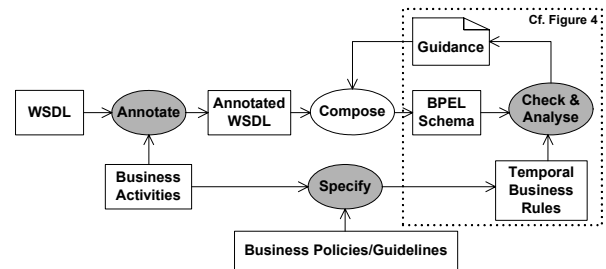


Figure 1: Overview of the Framework

that the BPEL designer can make use of the WSDL files as the basis for composition. The mapping is defined using the WSDL-S [3] semantic extension element *wssem:modelReference*. Figure 2 shows a small excerpt of an annotated WSDL.

```

<portType name="Customer">
  .....
  <operation name="issueInvoice"
    wssem:modelReference="
      http://example.com/TBRules.owl#IssueInvoice">
    .....
  </operation>
  .....
</portType>
  
```

Figure 2: An excerpt of an annotated WSDL

Specification of Temporal Business Rules. A valid or correct service composition for a domain must conform to a particular set of business rules. The type of the rules in a domain may vary. Our framework is particularly concerned with the *temporal* business rules, which constrain the occurrences or sequencing order of business activities. Since the temporal rules are stated at the business activity level, the framework incorporates a pattern-based language PROPOLS [17] for rule specification. PROPOLS is developed in the Web Ontology Language (OWL). It is built on and extends the Property Pattern System of [7, 8], and provides patterns as high-level abstractions of frequently used temporal logic formulas for specifying behavioural properties of Web service based applications. The patterns enable non-experts in formal languages to read and write formal property specifications. Below, we describe the key features of PROPOLS relevant to this work.

Figure 3 shows the basic constructs supported by [7, 8] and PROPOLS for property specification. The constructs on the left are temporal logic patterns and those on the right are scopes. A pattern specifies *what* must occur and a scope specifies *when* the pattern must hold. The *P*, *Q*, *R*, and *S* in the figure denote events (or business activities in this work) and *n* is a natural.

More specifically, scope “**globally**” refers to the whole running period of an application. Scope

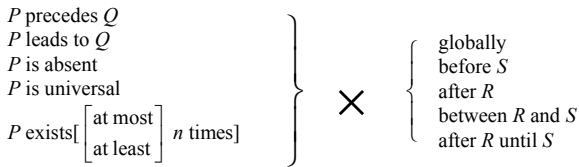


Figure 3: Constructs for Property Specification

“before S ” refers to the portion before the first occurrence of S . “after R ” refers to the portion after the first occurrence of R . “between R and S ” refers to the portion whose beginning is marked by R and end is marked by S . “after R until S ” is similar except that the portion can be open-ended and valid even without the occurrence of S .

For a given scope, pattern “ P precedes Q ” requires that the occurrence of Q always be preceded by the occurrence of P within the scope. Pattern “ P leads to Q ” requires that the occurrence of P always be followed by the occurrence of Q within the given scope. “ P is absent” states that P never occurs within the scope. “ P is universal” means that only P can occur within the scope. Finally, “ P exists” requires that P must occur within the scope, and it can be quantified with a minimum, maximum, or exact number of P ’s occurrences.

Domain experts could use these pattern and scope constructs to specify the temporal rules at the business activity level.

Guidance Generation. In the process of designing a complex BPEL schema which is subject to assorted temporal rules, a designer is always concerned with two questions: is the current design valid? If not, where are the problems and how to correct them? Our framework provides answers to the above questions by generating on-the-fly guidance throughout the design process. Basically, the framework offers two types of guidance: the *Proactive Guidance* that proposes suggestions on the next valid steps, and the *Remedy Guidance* that provides remedy suggestions for violations. Figure 4 below outlines the process of generating both types of guidance.

Before generating the guidance, the framework translates the BPEL schema into a FSA-based representation, where each transition is labelled with a business activity. In this process, the annotated WSDL files produced earlier are used to map service operation invocations in the BPEL schema to business activities. Additionally, the framework also translates each temporal business rule into a FSA that encodes the acceptable occurrence and sequence pattern of business activities. The translation is based on the semantics of the patterns and scopes predefined in

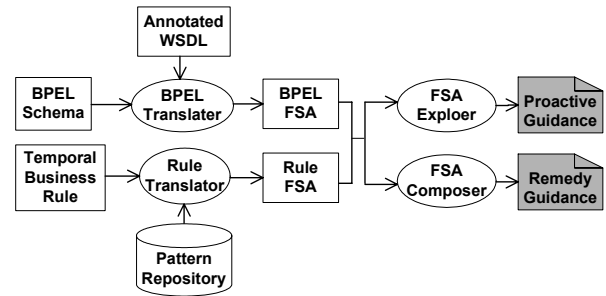


Figure 4: Process of Guidance Provision

PROPOLS.

The framework relies on the BPEL and rules FSA to generate guidance. For proactive guidance, the framework first explores the BPEL FSA to find out all the paths to the activity of concern. Then it appends the candidate activities to the end of each path and picks out the ones that will not cause violations to the rules as the valid “next step” activities.

For remedy guidance, the framework first checks the conformance of the BPEL schema to the temporal business rules by combining the BPEL FSA and rule FSA into a *composed FSA* using the techniques and tool developed in [17]. If a rule violation is detected, the framework analyses the checking results to identify the activities causing it, and suggests remedies to the BPEL composition. Currently, the framework employs three strategies to remedy an invalid BPEL schema: *activity insertion*, *activity deletion* and *activity re-ordering*. For each strategy, we have developed a generic algorithm that can generate remedy suggestions for any type of violations. The algorithms are based on the automatic state space exploration of the rule FSA and BPEL schema FSA.

A detailed explanation of the algorithms for guidance generation is given in section 3. For more details on conformance checking and the specification of temporal business rules (in the PROPOLS language), please refer to [17].

An Example Scenario. Let us consider an online order processing centre. The centre receives orders from customers and finds appropriate manufactures to fulfill the order. For its financial security, the centre stipulates two temporal rules for the order processing practice: (1) once the centre issues an invoice to the customer, a confirmation from the bank, which acknowledges the full payment made by the customer, must be received before the process proceeds; and (2) for the assurance purposes, the customer must pay a deposit before his/her order can be fulfilled. These two business rules can be expressed in PROPOLS as follows:

Rule 1: Customer.IssueInvoice leads to Bank.ConfirmPayment globally

Rule 2: Bank.ConfirmDeposit precedes Manufacturer.FulFilOrder before Bank.ConfirmPayment

These rules are represented as FSA in Figure 5. The circles denote states with double circles being final states, and the labeled arcs indicate state transition triggered by service operation invocations. A sequence of service operation invocations is considered as a rule violation if it ends the FSA at a non-final state. The mapping between the FSA events in the figure and business activities is given in Table 1.

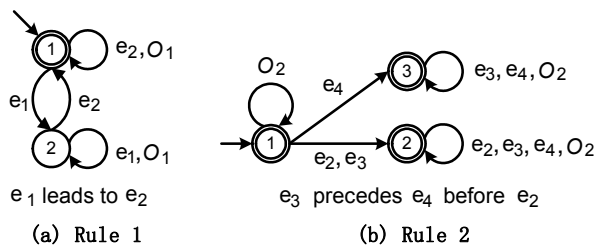


Figure 5: FSA representations of rules 1 and 2

e ₁	Customer.IssueInvoice
e ₂	Bank.ConfirmPayment
O ₁	Other business activities except e ₁ and e ₂
e ₃	Bank.ConfirmDeposit
e ₄	Manufacturer.FulFilOrder
O ₂	Other business activities except e ₂ , e ₃ and e ₄

Table 1: Mapping between FSA Events and Business Activities

Figure 6 shows an example invalid BPEL schema for the order processing centre, which violates Rule 1. Figure 6(a) is the graphic representation of the BPEL schema, where the shadowed activity are the one causing the violations. Figure 6(b) is the corresponding FSA, where the shadowed states indicate the error path (explained later in section 3.2).

3. Guidance Generation

In this section, we will elaborate the algorithms and strategies for generating proactive and remedy guidance, and demonstrate how they are used to correct the invalid BPEL schema shown in Figure 6.

3.1 Proactive guidance generation

The proactive guidance is the first line of defense that prevents a BPEL schema from violating business rules. Based on the current selected activity in a composition, this approach will proactively list the set

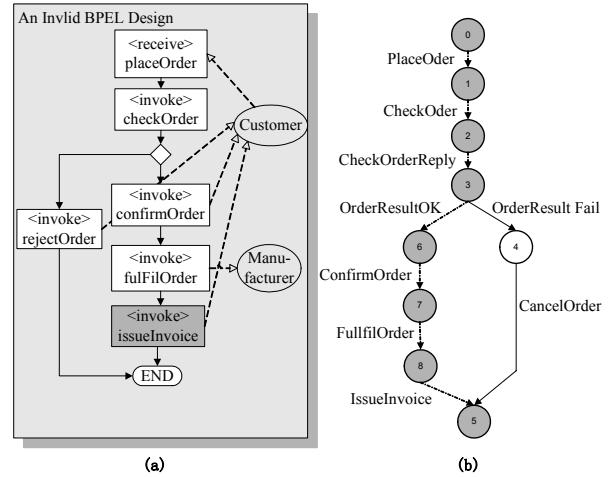


Figure 6: An Invalid BPEL Schema and its FSA

of valid “next step” business activities for the BPEL designer to choose from. A valid “next step” activity is the one that, if inserted by the designer after the currently selected position, will not violate any of the temporal business rules. The proactive guidance can be considered as a filter that eliminates the invalid activities from all the available ones at a given point. When designing a service composition that is subject to a rich set of complex temporal rules, the advantage of this approach becomes significant because it can dramatically narrow down the activities to choose from and hence improve the efficiency of the design process. The algorithm of proactive guidance generation is presented below in pseudo-code in Figure 7.

```

1 function getValidNextActivities(processFSA,
   ruleFSASet, currentSelection)
   as Set<BusinessActivity>
2 {
3   allPaths=findPath(processFSA,
4   processFSAInitialPoint, currentSelection);
5   for each pathk in allPaths{
6     ruleFSASetk=ruleFSASet.clone();
7     for each activityn in pathk{
8       for each ruleFSAj in ruleFSASetk{
9         ruleFSAj.advanceState(activityn);
10      }
11    }
12  PathValidActivitySet=findIntersection(
13  among all ruleFSA.getAcceptedEvent());
14  ValidNextActivitySet=findIntersection(
15  among all PathValidActivitySet);
16  return ValidNextActivitySet;
17 }

```

Figure 7: Algorithm for Proactive Guidance

This function is responsible for calculating and returning the set of valid “next step” activities (out-parameter *Set<BusinessActivity>*). It takes three inputs: the first one is the FSA representation of a

partially complete service composition (in-parameter *processFSA*), the second one is a set of rule FSAs (in-parameter *ruleFSASet*), and the last one is the current active point of the composition that the designer is working on (in-parameter *currentSelection*). The algorithm first calculates all paths (*allPaths*) between the initial point of the composition and the active point using the graph path-searching technique (line 2). For each path, it then instantiates a collection of FSA instances representing the temporal business rules (line 3, 4). These FSA instances will be used to check the validity of each path. Further, it updates the current states of the rule FSAs according to the transitions along the path (line 7). The intersection of all the accepted events of all rules FSAs at their current states would be the set of valid “next steps” along the path (line 11). The intersection of those sets of valid next steps along all the paths would be the set of valid expansion from the current point of the composition (line 12, 13). If the final result is empty then it means no further expansion is allowable from the current point of the composition.

3.2 Remedy Guidance Generation

When a rule violation is detected in the conformance checking, the framework starts an analysis process to identify the cause of the violation so that it can work out the guidance on remedies. More specifically, the framework first examines and compares the rules FSA, BPEL FSA and composed FSA to locate the error-entry activities (the activities causing the violations), then it uses customized path searching algorithms to explore the BPEL FSA to determine the error paths. An error path is a path or a sequence of activities in the BPEL FSA, which contains the error-entry activities that lead a rule FSA to an error state. Generally there are two possible cases that can result in an error path: (1) a path causes the rule FSA to raise an exception or rejection, because some activities in the path are unspecified or unacceptable at the corresponding FSA state; or (2) a path causes the rule FSA to end at a non-final state [14]. The invalid BPEL schema shown in Figure 6(a) is an example of case 2. Its error-entry activity *Customer.IssueInvoice* leads the FSA of rule 2 to a non-final state (*state 2*), hence causing a violation.

Once the error paths are identified, the framework starts to use various strategies and algorithms to generate different types of remedy suggestions. Currently, the framework adopts three strategies for remedy generation: *Activity Insertion* – insert one or more activities in the error path to correct it, the insertion position is either before or after the error-entry activity; *Activity Deletion* – removing one or more activities, which may include the error-entry activities, in the error path; and *Activity Re-Ordering* – change the

sequencing order of some activities that are related to the error-entry activity in the error path. It is worth noting that all these strategies are only concerned with the activities that are of interest to the violated rule. That is, the framework only inserts, deletes or re-orders the activities whose occurrence may cause a state change or a rejection at some state in the rule FSA. The other activities will be left unchanged.

For each strategy, we have developed a generic algorithm that can generate remedy suggestions for any type of rule violations. Given an error path identified in the violation analysis step, each algorithm explores the violated rule FSA and attempts to find out a solution to correct the error. It is worth noting that for complex BPEL schemas, the size of possible remedies can be very large. Therefore we restrict the algorithms to only seeking the remedies that insert, remove, or change activities at the closest position to the error-entry activities to avoid the scalability problems. The algorithms for each strategy are explained below:

Activity Insertion. This strategy tries to insert one or more activities in the error path of a BPEL schema to remedy its violations. It consists of two sub-strategies: *Forward Extension* and *Backward Insertion*, indicating different preferences on insertion positions.

The *Forward Extension* strategy attempts to find the shortest sequence of business activities that, if added to the end of the schema, can make the BPEL FSA conform to the violated rule. The remedy suggestions of this strategy will not change the existing BPEL schema design.

Figure 8 illustrates the algorithm for the *Forward Extension* strategy in pseudo-code. Essentially, it is used to find a sequence of events that can bring the violated rule FSA from the current non-final state to a final state. Inserting this particular sequence of events in the end of BPEL schema would make it conform to the previous-violated rule. Technically, the algorithm uses the depth-first search approach to search for a path from a given “start” state to any “target” state. It recursively traverses all the paths leading out of the “start” state to seek the “target” state, and returns the first path found. When applying the algorithm in generating forward extension remedies, the “start” state is set as the last state of the rule FSA before it becomes violated, and the “target” state is any final state.

Backward Insertion is applied when a user misses a sequence of activities in the BPEL design. This strategy attempts identifying the missing steps by *backward* exploring the error path from the error state, and consulting the violated rule FSA at the same time to find what activities to insert and where to insert. Figure 9 illustrates the algorithm for this strategy.

This algorithm starts with *findPreviousMissingSteps*.

```

function findPathToFinalState(startState,
                             fsa, excludedStates) as path
{
  if not fsa.contains(startState)
  then return null;
  for each nextState in
    ruleFSA.getNextStates(startState) {
    if nextState in excludedStates
    then continue;
    else if nextState is finalState then
      return nextState;
    else{
      excludedStates.add(nextState);
      path=findPathToFinalState(nextState,
                               ruleFSA,excludedStates);
    }
    if path!=null then return nextState+path;
  }
  return null
}

```

Figure 8: Algorithm for Forward Extension Strategy

The latter *missingStepsAfterState* is a recursive function that, given a current state (*cState*) of a rule FSA and an error path (*errorPath*), returns a sequence of activities and the location to insert. In each round, it backtracks a step along the *errorPath*, and examines the missing steps between *cState* and its previous state (*pState*) by calling *missingStepsAfterState*. The latter function examines if the sequence fragment of *errorPath* after *pState* is acceptable at any state reachable from *pState*, and returns an activity sequence between *pState* and that state. The set *excludedStates* is used to prevent infinite execution loops. Function *isActSeqAcceptedFromState* tests if a given sequence leads the rule FSA from the given state to a final state.

Activity Deletion. This strategy takes a rather straightforward approach. It attempts to correct a BPEL schema by simply deleting the error-entry activity that causes a violation. For certain type of rule violations, such as those violate pattern *exists* or *exists at most*, it is the best or probably the only appropriate strategy, because it will suggest the user to delete the activities that exceed the maximum occurrence. For other type of rules, however, it may be user's last choice, since it tries to remedy the violation in a "destructive" manner.

Activity Re-Ordering. This is a conceptually simple but rather tricky strategy. It is based on the following philosophy: a violation at the pattern level will not be considered as a violation to the rule if it is not in the proper scope. Therefore this strategy attempts to eliminate the rule violation by taking the error-entry activity out of the applicable scope through re-ordering. The advantage of this strategy is that it could make an incorrect BPEL schema become valid without inserting or removing any activities in the current design.

Due to space limitation and their simplicity, the

```

function findPreviousMissingSteps(cState,
                                 errorPath, ruleFSA)
  as(cState, SeqToInsert){
  if (not ruleFSA.contains(cState) or
      cState.isInitial()) return null;
  pState = ruleFSA.getPreviousStateByPath(
    cState, errorPath);
  postActSeq=ruleFSA.getActSeqAfterStateByPath(
    pState, errorPath);
  actSeqToIns=missingStepsAfterState(pState,
    postActSeq, ruleFSA, emptySet);
  if (actSeqToIns != null)
    return (cState, actSeqToIns);
  else return findPreviousMissingSteps(pState,
    errorPath, ruleFSA);
}

function missingStepsAfterState(cState, actSeq,
                               ruleFSA, excludedStates)
  as eventSequence{
  if (ruleFSA.isActSeqAcceptedFromState(cState,
    actSeq)) return emptySet;
  excludedStates.add(cState);
  for each (nEvent, nState) in
    ruleFSA.getNextEventStatePairs(cState){
    if (nState in excludedStates) continue;
    if (ruleFSA.isActSeqAcceptedFromState(
      nState, actSeq)) return nEvent;
    else
    {
      actSeqToIns = missingStepsAfterState(
        nState, actSeq, ruleFSA);
      if (actSeqToIns != null)
        return nEvent + actSeqToIns;
    }
  }
  return null;
}

```

Figure 9: Algorithm for Backward Insertion Strategy

algorithms for the *Activity Deletion* and *Activity Re-Ordering* strategies are omitted here.

It is worth noting that when a rule violation is detected, the framework will use all the available strategies to generate remedy suggestions. It is up to the designer to choose the most appropriate remedy suggestion from multiple options to meet the business requirements.

When applying these strategies to remedy the invalid BPEL schema shown in Figure 6(a), it takes two steps to correct it. Since the schema violates the rule 1, the framework generates two remedy suggestions: one is from the *Forward Extension* strategy, which suggests inserting *Bank.ConfirmPayment* at the end of the schema, the other is from *Activity Deletion*, which suggests removing the *Customer.IssueInvoice*. The designer chooses the insertion instead of deletion and it brings the FSA of rule 1 from the non-final state (*state 2*) to the final state (*state 1*), and therefore eliminates the violation to rule 1. After *Bank.ConfirmPayment* is inserted, another violation to rules 2 is detected because the activity *Manufacturer.FulFilOrder* leads the FSA of rule 2 to *state 3*, which rejects the following activity *Bank.ConfirmPayment*. Then the framework proposes another two remedy suggestion: one is from the *Back Insertion* strategy, which suggests inserting

Bank.ConfirmDeposit before *Manufacturer.FulFilOrder*, and the other is simply deleting *Bank.ConfirmPayment*. The designer adopts the *Back Insertion* strategy and remedies the violation of rule 2. Figure 10 (a) and (b) illustrates the changes of BPEL design during this remedy process, where the shadowed activities are newly inserted following the guidance.

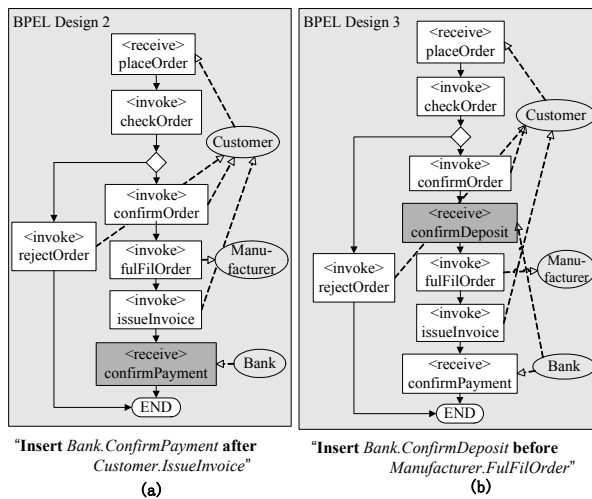


Figure 10: Remedied BPEL Designs

4. Implementation

We have developed a prototype of the guidance framework, called *BPELGuide*, in Java 5.0. *BPELGuide* employs the *BPEL2FSA* translator developed in [17] to convert BPEL schema to a FSA, and also the checking engine in [17] to check the conformance of BPEL FSAs to rule FSAs. During the BPEL composition process, *BPELGuide* monitors a given BPEL schema file, which may be produced by a BPEL composition tool, e.g. *ActiveBPEL Designer* [1]. Whenever a newer version of syntactically correct BPEL schema is saved, *BPELGuide* invokes *BPEL2FSA2* to obtain the schema FSA and passes it on to the checking engine for conformance testing against each given temporal business rule. When a rule violation is detected, it identifies the error paths and associated error-entry activities and automatically generates remedy suggestions.

For illustration purposes, Figure 11 shows a screenshot of the prototype working in the *ActiveBPEL Designer* environment. The *Guidance* window displays the automatically generated guidance for the *BPEL Design 2* illustrated in Figure 10 (a). It explicitly informs the designer the problem of the current schema (*Violated Rule*), the cause of the problem (*Error Path* and *Error Entry Activity*) and how to correct it (*Remedy Plans*). Furthermore, by clicking the "Show Details"

button, the designer can see the graphical representation of the BPEL FSA, where the error path and the location of the insertion or deletion are highlighted. Information like this offers great help during the process of service composition.

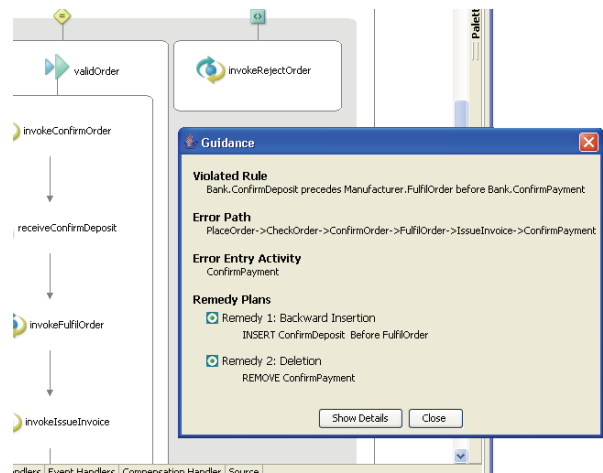


Figure 11: A Screenshot of the Prototype Working in the *ActiveBPEL Designer* Environment

5. Related Work and Discussion

A body of work exists in the general area of service composition. [13] proposes an approach to automatically generating dynamic service compositions according to a set of business rules. [11] employs the AI planning techniques to generate executable composite processes to achieve the given goal. [6] uses FSMs to model the external behaviours of both component services and composite services and checks the existence of a composition model as a satisfiability problem in DPDL Logic. [12] uses situation calculus and Petri nets to define the semantics of Web services described in DAML-S to enable sequential service composition.

Some approaches particularly target at the BPEL design. [5, 10] adopt the model-driven architecture to automatically generate BPEL code skeletons from abstract process models that are manually built in UML activity diagrams. [15, 16] utilise rich semantic descriptions of Web services or abstract BPEL processes and model checking based planning techniques to automatically produce executable BPEL processes.

The approaches mentioned above put focus on the automatic generation of service composition. In general, automatic service composition may only be possible in the well-defined and well-constrained problem domains. However, most of real-world problems are far more

complex and require human input to reach a valid service composition as a solution. Our approach recognises the necessity of human intervention and provides decision making support to the BPEL designers by generating on-the-fly guidance during the service composition process. This can bring immediate benefits to the current practice.

To our knowledge, there has not been any approach that proactively provides automated in-process composition guidance in terms of determining the next valid steps, identifying missing and misplaced activities, and suggesting corrective actions.

6. Conclusion

Providing guidance during the application development process is an intricate problem not only in the context of service composition, but also in the context of general software development. However, it has immediate benefits to the current day-to-day practice of service-oriented application development, in terms of well-advised business process construction and just-in-time error detection and correction.

In this paper, we have presented a framework and associated techniques to guide the human-driven process of service composition according to temporal business rules. Two kinds of guidance, proactive and remedy guidance can be generated by the framework and both of them adopt various strategies and generic algorithms that uses FSA exploration techniques to automatically generate guiding suggestions.

Currently, we are optimising the implementation of the framework by improving its integration into the *ActiveBPEL Designer* to present more comprehensive guidance information in the IDE environment. We are also investigating the possibility of extending the framework to cater for other types of business rules, e.g. those in [13].

References

- [1] "ActiveBPEL Designer", <http://www.active-endpoints.com/products/activebpedes/>
- [2] BPML. Business Process Modeling Language", <http://www.bpmi.org/>, 2002
- [3] R. Akkiraju, J. Farrell, and *e. al.*, "WSDL-S", <http://www.w3.org/Submission/WSDL-S/>
- [4] T. Andrews, F. Curbera, H. Dholakia, Y. Golland and *et al.*, "Business Process Execution Language for Web Services version 1.1", <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>
- [5] R. Anzböck and S. Dustdar, "Semi-automatic generation of Web services and BPEL processes - A Model-Driven approach", Proc. *3rd Int'l Conference on Business Process Management (BPM)*, 2005, pp. 64-79, Springer.
- [6] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini and *et al.*, "Automatic composition of e-services that export their behavior", Proc. *1st Int'l Conference on Service Oriented Computing (ICSOC)*, Trento, Italy, 2003, pp. 43-58, Springer.
- [7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-state Verification", Proc. *International Conference on Software Engineering*, 1999, pp. 411-420.
- [8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property Specification Patterns for Finite-State Verification", Proc. *Workshop on Formal Methods in Software Practice*, 1998, pp. 7-15.
- [9] J. Han, Y. Han, Y. Jin, J. Wang and *et al.*, "Personalized Active Service Spaces for End-User Service Composition", Proc. *the IEEE International Conference on Services Computing (SCC)*, Chicago, USA, 2006, pp. 198-205, IEEE Computer Society.
- [10] K. Mantell, "From UML to BPEL", IBM developerworks, <http://www-128.ibm.com/developerworks/webservices/library/ws-uml2bpel/>, 2003
- [11] S. McIlraith and T. C. Son, "Adapting Golog for Composition of Semantic Web Services", Proc. *8 International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Toulouse, France, 2002, pp. 482-496.
- [12] S. Narayanan and S. A. McIlraith, "Simulation, verification and automated composition of web services", Proc. *11th international conference on World Wide Web (WWW)*, Honolulu, Hawaii, USA, 2002, pp. 77-88, ACM Press.
- [13] B. Orriens, J. Yang, and M. P. Papazoglou, "Model Driven Service Composition", Proc. *1st International Conference on Service Oriented Computing (ICSOC)*, Trento, Italy, 2003, pp. 75-90, Springer-Verlag.
- [14] M. T. Phan, "Using Temporal Business Rules to Verify and Guide Service Composition," *Faculty of Information and Communication Technologies*, Bachelor of Information Systems (Honours), Swinburne University of Technology, 2006.
- [15] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso, "Automated Composition of Web Services by Planning at the Knowledge Level", Proc. *Internal Joint Conference on Artificial Intelligence (IJCAI)*, Edinburgh, Scotland, 2005, pp. 1252-1259.
- [16] P. Traverso and M. Pistore, "Automated Composition of Semantic Web Services into Executable Processes", Proc. *3rd International Semantic Web Conference (ISWC)*, Hiroshima, Japan, 2004, pp. 380-394, Springer.
- [17] J. Yu, T. P. Manh, J. Han, Y. Jin and *et al.*, "Pattern Based Property Specification and Verification for Service Composition", Proc. *7th International Conference on Web Information Systems Engineering (WISE)*, Wuhan, China, 2006, pp. 156-168, Springer.