An Algorithm for the Validation of Executable Completions of an Abstract BPEL Process

Ralph Mietzner, Zhilei Ma, Frank Leymann

Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstr. 38
70569 Stuttgart, Germany
{mietzner,ma,leymann}@iaas.uni-stuttgart.de

Abstract: WS-BPEL is the standard for specifying and executing business processes by orchestrating Web Services. Abstract and executable processes are two kinds of BPEL processes that are defined in the BPEL standard. An abstract process can be used as a process template, which can be completed and made executable through "executable completion". The BPEL standard defines a set of rules that must be obeyed during such an executable completion. In this paper, we present an algorithm for validating whether an executable BPEL process is a valid executable completion of an abstract BPEL process. Our approach advances the existing XML comparison algorithms in a way that it takes the BPEL-specific characteristics into account and is optimized towards the validation of "executable completion" of abstract BPEL processes.

1 Introduction

WS-BPEL 2.0 [BPEL07] (BPEL for short) defines a collection of constructs for modelling abstract and executable business processes. An abstract BPEL process is a partially defined business process that hides some internal operational details needed for the execution. Operational details can be hidden in abstract BPEL processes by marking them as opaque or by simply leaving them out by using omission shortcuts. The marked operational details are called opaque tokens. In general, opaque tokens are allowed for all attributes and expressions present in elements of the abstract process. Furthermore, an activity as a whole can be marked as opaque by using opaque activities. Through "executable completion", an abstract process can be transformed into an executable process by resolving the opaque tokens in the abstract process. In addition to that, an abstract process can also be enriched by adding any further BPEL process elements during executable completion, i.e. the abstract process may be extended during executable completions by elements not foreseen in the abstract process.

The BPEL standard defines two usage profiles for abstract processes. A usage profile specifies a particular use of the concept of an abstract process by defining constrains for executable completion. There are two typical use cases of the concept of abstract processes: The first one is for describing the observable behaviour of an executable process, the second one is for describing essential process logic excluding concrete execution details. Observable behaviour of a process is typically used as guideline for interacting with the underlying executable process; in this sense, a corresponding abstract process defines a "business protocol". Essential process logic of a process is typically used to ensure that a certain executable process implements the essential logic; in this sense a corresponding abstract process defines a "template".

In a Software as a Service (SaaS) scenario a provider could define a template process, which can be customized at certain positions to satisfy individual requirements of each customer. For example, a SaaS provider defines a “Abstract Travel Request Process” as shown in Figure 1, which specifies the basic processing steps for a travel request, such as “request the travel details”, “invoke the travel agency”, and “send out the itinerary”. Additionally, the template process contains also an opaque activity, which allows the customization of the template process. Figure 1 also shows three executable completions of the template process. In the first executable completion, the customer completes the abstract process with an activity “Ask Manager for Conformation”. In the second executable completion, the customer replaced the opaque activity with an empty activity denoting that no additional step is needed. The third executable completion is invalid, since the customer added an activity (the “Charge Customer” activity) at a point in the abstract process where no opaque activity exists, which violates the executable completion rules defined in the template profile for abstract processes. [BPEL07].

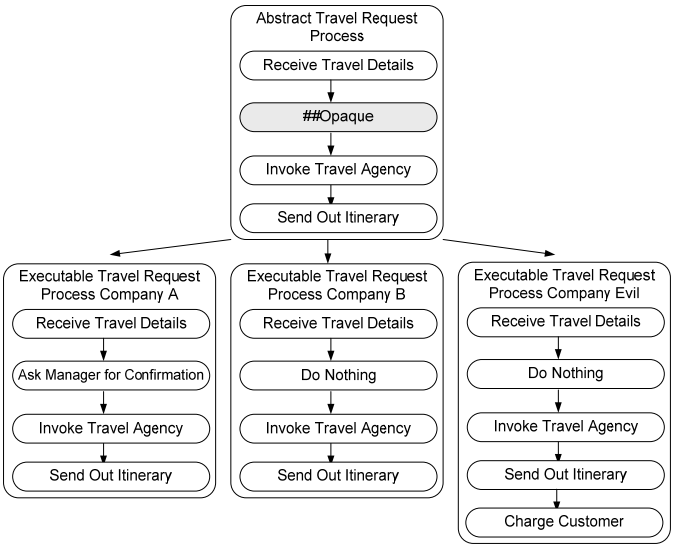

Figure 1: An example of an abstract process and its three executable completions.

In this paper we present an algorithm for validating whether an executable BPEL process is a valid executable completion of an abstract BPEL process. We analyze the BPEL-specific characteristics, which lead to the conclusion that existing algorithms for the comparison of XML documents are not sufficient for determining whether an executable BPEL process is a valid executable completion of an abstract BPEL process. Our algorithm takes the BPEL-specific characteristics into account and has an upper bound of $O(N_e^{5/2})$ in the worst case with N_e being the amount of nodes in the executable BPEL process.

The remainder of this paper is organized as following: In Section 2, we analyze the BPEL-specific characteristics that must be considered explicitly during the validation of executable completions, and discuss why the existing work is not sufficient or efficient enough for this use case. Section 3 presents the algorithm in details and analyses the complexity of the algorithm step by step. Section 4 concludes this paper and presents our future work.

2 BPEL Characteristics and Related Work

Several BPEL characteristics need to be explicitly considered when dealing with the validation of the executable completion of an abstract BPEL process:

- *Unique identifiers do not exist.* Some BPEL elements, i.e., BPEL specific XML elements in the BPEL process document do not have unique identifiers. When considering BPEL activities, the attribute "`name`" is the only candidate for a unique identifier. However, this attribute is defined as optional in the BPEL standard and several activities in a process can have the same name. Therefore, it is in general not possible to compare a process element in an abstract process with a process element in an executable process based on the "`name`" attribute.

- *Children of a process element can be ordered or unordered.* In BPEL processes, the order of the child elements of a BPEL element, for example the child activities of an activity, is not necessarily important. BPEL activities enclosed in a `sequence` activity are by definition ordered. Furthermore, the `else` element must be placed as the last child of an `if` activity. In any other cases, the order of children elements is undefined. Thus, they may be presented in any arbitrary order without affecting the behavioural semantic of the BPEL process.

- *Deletion is forbidden, addition is allowed.* During executable completion, it is solely allowed to modify or add certain BPEL elements, while deleting BPEL elements is forbidden. Thus all process elements and their attributes presented in the abstract BPEL process must also be presented in the corresponding executable BPEL process. Nevertheless, the values of the attributes might have been modified or new child elements might have been added.

- *Existing parent-child relations must not be modified.* Rules defined in the BPEL standard forbid changing the parent of a BPEL element during executable completion. In other words, the existing parent-child relations of BPEL elements must not be modified.

- *Opaque tokens can exist in the abstract process.* Opaque tokens in abstract processes lead to situations where several elements of an executable process are valid executable completions of the same element in an abstract process. Figure 2 shows a part of an executable and an abstract process represented in a tree structure, which illustrates the problem of opaque tokens and unordered children. Since the first variable declaration in the abstract process consists only of opaque tokens, it can be mapped to each of the three variables in the executable process on the left side. Therefore, it can not be determined which variable in the executable process is an executable completion of the opaque token.

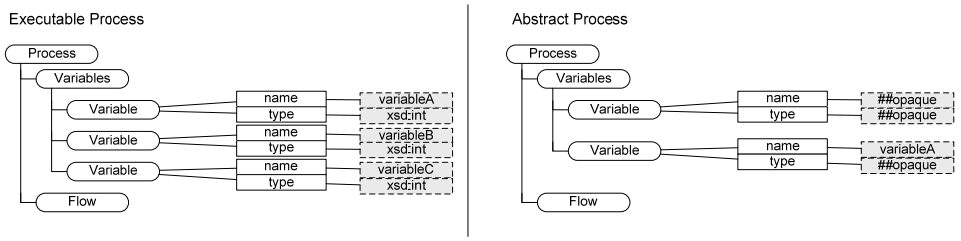

Figure 2: Parts of an executable and abstract process.

Other work, such as [Mar05b] [Mar05a], compares abstract and executable processes based on the incoming and outgoing messages or operating guidelines [MW06]. These approaches have a different focus than ours and are particularly useful when searching for partner processes that follow a certain interaction pattern. However, the comparison does not take the internal process logic into consideration. Since BPEL is an XML-based language, a BPEL process definition is an XML tree. A lot of research has been done for comparing text files and the results have been incorporated in many tools. The GNU diff [GNU] tool is one prominent example of using algorithms, e.g. the LCS [Mye86] (Longest Common Subsequence) algorithm, for comparing flat files. However, Chawathe et al. [CRGMW96] pointed out that these algorithms do not work particularly well with hierarchical data such as SGML and XML, because they do not take the information embodied in the hierarchical structure of an XML document into consideration. Several other approaches ([CAM],[CAM02] [WDC03]) leverage the fact that all XML documents can be represented as trees and utilize tree comparison algorithms to compute the edit distance of them. An edit distance of two trees indicates the necessary modification steps to derive the second tree from the first one. Using a cost function for modification of a tree, a minimal edit distance is defined as the least expensive set of edit operations needed for transforming one tree into another. But [CAM] does not take unordered children into account and can not compute the minimal edit distance. [WDC03] matches the nodes and sub-trees in the two XML documents based on a hash function. This approach is not feasible in the BPEL case, since the sub-tree of an abstract process containing opaque tokens has a different hash value than a

possible executable completion. [ZSS92] stated that the computation of an edit distance of trees with unordered children is MAX SNP-hard [ZJ94]. However, the complexity can be reduced to quasi quadratic complexity by constraining the possible mappings between nodes of the same parents [Zha96]. This constraint is satisfied in executable completions, since parent-child relations must not be modified. In [Val01], an algorithm for computing bottom-up distance between trees is presented, which also works with unordered trees and runs in $O(n)$. Instead of computing the minimal edit distance between two trees, we verify that BPEL specific XML elements have not been added or removed at places where adding or removing are not allowed during the executable completion of an abstract process. The use of opaque tokens in the abstract process imposes special problems for the match-making of process elements, since the algorithm needs to determine which process element in the executable process is an executable completion of an opaque token in the abstract process. As Figure 2 depicts, under certain circumstances, one process element in an abstract process can be matched to more than one process elements in an executable process. Furthermore, an executable process always differs from the corresponding abstract process, since at least the namespace of the abstract process has to be changed to that of executable BPEL. Therefore the usage of general tree comparison algorithms is not sufficient for our use case.

3 The Algorithm

This section describes the comparison algorithm in more detail. The algorithm can be divided into four phases: the preparation phase, the comparison phase, the node identification phase and the node addition phase.

3.1 The Preparation Phase

The preparation phase reads the executable process document and stores all the nodes of the process tree in a data structure that can later be queried easily in the comparison phase. Since every BPEL artefact present in the abstract process must also be present in the executable process, we first built a data structure to store the artefacts of the executable process and then try to find each artefact of the abstract process in this data structure. The algorithm traverses the executable process document and creates a hash map for each XML element type or attribute type of the executable process document. Maps are created dynamically at runtime providing support for arbitrary extension elements such as the WS-BPEL extensions for people [B4P07]. The hash maps also contain the namespace so that the algorithm can distinguish similarly named elements and attributes from different XML namespaces.

Algorithm 1 works as follows: First, the XML element that is under examination is added as a (*key*, *parentKey*) tuple to the respective map corresponding to the XML element (i.e., the variable map, the partnerLink map, the flow map). The key item is a unique identifier for the element. The unique identifier is an integer that is increased by one for each element that is found. The *parentKey* is the key of the parent element.

Algorithm 1 Computing the element and attribute maps

```
procedure addElement(Element e, Integer parentKey)
    map = getMap(e.type)
    amountOfElements++
    key = amountOfElements
    map.add(key, parentKey)
    for all a in e.getAttributes() do
        map = getMap(e.type + a.type)
        parentKeyList = map.get(key)
        parentKeyList.add(parentKey)
        map.add(a.value, parentKeyList)
    end for
    for all c in e.getChildElements() do
        addElement(c,key)
    end for
end procedure
```

As the second step, the attributes of the element are examined. Each attribute is added to the respective map. There exists one map for each attribute type and element type combination, i.e., a name attribute declared at a flow element is added to a different map than a name attribute declared at a variable element. Each attribute map is filled in the following way: Each entry consists of a (*key*, *parentKeyList*) tuple. The key is a hash value of the value of the attribute and the *parentKeyList* is a list of keys of elements at which an attribute with the value "key" is declared in the executable BPEL process document. The child elements of the processed element have been examined in the same way.

The resulting hash maps for the running example are shown in Figure 3. The figure only shows the maps built for the part of the process shown on the left of the figure for simplicity reasons. However the figure shows all important aspects, namely how elements are stored and how attributes and their values are stored. Elements that are text-nodes are stored like attributes.

During this step the tree is traversed once and each node is added to the respective map. Therefore the complexity of this step is $O(N_e \times (1 + h_e))$, where N_e is the amount of nodes in the executable tree and h_e is the complexity of building the hash for a node and adding it to the respective map.

3.2 The Comparison Phase

Having examined the executable process, the algorithm can now do the actual comparison using the work that has been done in the preparation phase. Therefore, the algorithm traverses the abstract process tree depth-first. The comparison function takes as input a node (*abstractNode*) and a set of parents (*parentList*). The *abstractNode* is the

node of the abstract process for which we want to find the possible executable completions. The *parentList* is the set of parent nodes that are possible executable completions of the parent node of *abstractNode*. The output is all nodes of the executable process that are valid executable completions of *abstractNode* and children of the nodes in *possibleParents*.

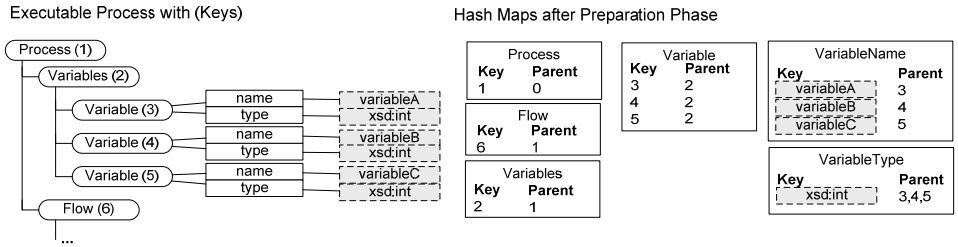

Figure 3: Maps of the example executable process after the preparation phase

As the first step, all keys of the (*key*, *parentkey*) values in the map corresponding to the type of *abstractNode* (i.e., `variable`, `flow`, `assign`,...) and that have a parent key entry that corresponds to one of the parents in *possibleParents* are returned and saved in the *possibleNodes* set. In the second step all attributes of *abstractNode* are compared. For each attribute of *abstractNode*, a lookup in the corresponding attribute map is performed with the hash value of the attribute. The set of *parents* keys returned is intersected with those returned in the first step and stored in the *possibleNodes* set. The *possibleNodes* set now contains all those executable elements that are of the correct type, having a similar parent and the same attributes as the abstract process element to be compared. The third step is now to assure that the child nodes of all the nodes in *possibleNodes* are valid executable completions of *abstractNode* too. For that reason, the function *compare* is now called recursively with all child nodes and the *possibleNodes* set. The results of each call of the function *compare* are intersected with the *possibleNodes* set. The function then annotates the *abstractNode* with the *possibleNodes* set and returns the *parentKeys* for each *possibleNode*. In case there are no elements in the executable process that correspond to an element in the abstract process, the algorithm can mark the element with an error, indicating that the executable BPEL is no valid executable completion of the abstract BPEL.

Ordered Children

When comparing abstract and executable BPEL we cannot always assume that child nodes are unordered. For example if the parent element of a set of children is a `sequence` activity or an `if` activity, the order in which the child elements appear matters. In these cases an element that appears before another element in the executable process must also appear before the corresponding element in the abstract process. Assuring the ordering is fairly simple, since we can use the fact on how we distributed the keys for the elements in the preparation phase. Since we traversed the tree in depth-first manner, the element with the lower key of two elements with the same parents is the element which has been declared before the other one in the executable process.