Verifying Business Rules Using an SMT Solver for BPEL Processes

Ganna Monakova[1], Oliver Kopp[1], Frank Leymann[1], Simon Moser[2], Klaus Schäfers[2]
[1]Institute of Architecture of Application Systems, Stuttgart, Germany
[2]IBM Deutschland Research & Development GmbH, Böblingen, Germany
[1]lastname@iaas.uni-stuttgart.de [2]{smoser|kschaef}@de.ibm.com

Abstract: WS-BPEL is the standard for modelling executable business processes. Recently, verification of BPEL processes has been an important topic in the research community. While most of the existing approaches for BPEL process verification merely consider control-flow based analysis, some actually consider data-flows, but only in a very restrictive manner. In this paper, we present a novel approach that combines control-flow analysis and data-flow analysis, producing a logical representation of a process model. This logical representation captures the relations between process variables and execution paths that allow properties to be verified using Satisfiability Modulo Theory (SMT) solvers under constraints represented by the modelled assertions.

1 Introduction

Many of today's enterprises model their business processes in BPEL [OAS07]. In order to ensure quality of the modelled process, its correctness has to be proved. A correct business process always terminates and produces valid results. Successful termination implies the absence of deadlocks and can be verified using several techniques, e.g. [Hol04, MM06, Loh07]. These techniques merely consider the control-flow of the process and abstract from the data-flow. A valid result for a business process is usually defined by business constraints on the produced output, e.g. "A customer who ordered less than 100 items should not receive a discount". The verification of such a business constraint depends on the relations between different process variables: to be able to verify the above constraint we need to know the relation between number of ordered items and calculated discount. Such relations depend on both control-flow and data-flow.

We use a simplified price calculation process that is a part of an onlineshop process to illustrate the concepts of our approach. The basic idea of the process presented in Figure 1 is to determine the price of an item depending on the amount of ordered items. A discount is given based on the subtotal. The process receives the number of ordered items n and determines the item price dependent on n. If the customer has ordered more than 100 items, they can buy each for 10 €, otherwise a price of 20 €per unit is offered. After the subtotal is calculated, the customer gets 10% discount if it is between 1.000 € and 2.000 € and 20% discount if the subtotal exceeds 2.000 €. The total sum s is sent back to the customer. The process contains both graph-based (`flow`) and block-structured (`if`) constructs to show

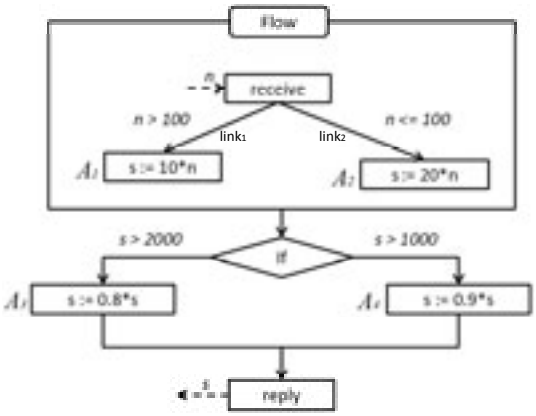

Figure 1: Price calculation process

that the presented approach is capable of handling both.

Assume that a business analyst wants to ensure that the modelled process satisfies the following business constraints: "A customer who ordered less than 200 items should not receive a discount of more than 10% " and "If the customer ordered 50 items or more, they should get a discount". To enable the verification of these constraints, the relations between the variables s and n have to be analysed: At first glance, the control flow decision made in the `if` activity considers the value of s only and does not depend on the decision made in the preceding `flow` activity. However, if the number of ordered items exceeds 100 ($n > 100$), then we know that A_1 will be executed, and thus $s = 10 * n > 10 * 100 = 1000$. With this knowledge, we can conclude that the `if` construct will never be skipped, meaning the customer will be assured of a discount. This analysis is only possible if we know the relation between s and n, namely $s = 10 * n$, and know the condition which enables this relation, namely $n > 100$. We also call such a condition the *Execution Condition (EC)* of activity A_1. Note that we do not make any assumptions on the values the variable n can take, but only analyse the connections between data conditions and data manipulations.

The above example shows that a relationship between variables enables restrictions on possible control flows to be analyzed. As an alternative example, the execution path $receive$, A_2, A_3, $reply$ is not possible: if n is 100, then the condition $s > 2.000$ evaluates to false, because s was assigned a value of 2.000 after A_2 occured.

In summary, the relationship between variables will have an impact on the evaluation of the conditions that drive the control-flow. For this reason they play an important role in the process analysis. In this paper, we present a technique that analyses these relations and models them using *logical assertions*. Logical assertions capture the execution semantics of a BPEL process and form the verification basis for the business constraints. The negation of a business constraint is added to the verification base and checked to ensure it is unsatisfiable. If it is unsatisfiable, then the business constraint is valid. Otherwise a counter example violating this constraint is found.

The satisfiability of the modelled assertions is checked using the Satisfiability Modulo Theories (SMT) solver Yices [DdM08]. An SMT solver solves satisfiability problems for

Boolean formulas containing predicates of underlying theories. Such theories can be, for example, theories of arrays, lists and strings [B+06]. In addition, an SMT solver can be extended with new theories as shown in [ND79]. To present the proposed approach, we consider the theory of the linear arithmetic.

The remainder of this work is organized as follows: Section 2 presents how the execution condition can be derived for each activity in a BPEL process. Section 3 shows how a BPEL process can be analysed and modelled with logical assertions. The verification of business constraints within the modelled context is presented in Section 4. Section 5 compares the presented approach with related work and 6 concludes and provides an outlook on future work.

2 Determining Execution Conditions

An execution condition of an activity is a Boolean expression that is constructed recursively by analysing all conditions that have to be satisfied to enable the execution of this activity. The following conditions have to be analysed:

1. A `flow` activity models concurrency. Additional synchronisation between activities can be modelled with links. Each activity in a flow can have an arbitrary number of incoming and outgoing links. Each link has exactly one source and one target activity and expresses the synchronisation dependency between them. Each link has a transition condition, which is evaluated if the source activity was successfully executed. The transition condition is an arbitrary[1] XPath expression of return type $Boolean$.

   If a transition condition is not explicitly defined, the link gets a default transition condition $true$. Each link has a status that can be either *unknown*, *negative* or *positive*. A link status can become positive if and only if the source activity of the link was successfully executed and the transition condition of the link evaluates to $true$. If the source activity was skipped or the transition condition evaluates to false, the link status becomes negative. This is called *dead path elimination (DPE)*. More information regarding `flow` semantics and DPE is given in [OAS07, C+03]. Each activity in a `flow` has a join condition. A join condition is a Boolean function over the status values of the incoming links. The default join condition is a disjunction of all incoming link status values.

   According to the flow semantics described above, the execution condition (EC) of an activity A can be recursively constructed as follows:

$$\begin{aligned} EC(A) &= JC(A) \\ JC(A) &= f(S(L_1)...S(L_n)) \\ \forall i \in [1..n] \ : \ S(L_i) &= L_i.tc \wedge EC(L_i.source), \text{ where} \end{aligned}$$

[1] Recall that in this work we use linear arithmetic theory and therefore consider linear arithmetic expressions only

A denotes an activity in a `flow` with L_1,..., L_n incoming links (or rather A is the target of L_1,..., L_n), $JC(A)$ denotes the join condition of the A, $L_i.tc$ denotes the transition condition of the link L_i, $L_i.source$ denotes the source activity of the link L_i, $S(L_i)$ denotes the status of the link L_i and *f(. . .)* is an arbitrary Boolean function, which specifies the join condition [OAS07].

In example of Figure 1 the `flow` activity contains three activities. The `receive` activity does not have any incoming links, therefore both its join and execution condition are true. Activities A_1 and A_2 have one incoming link and the default join condition. Therefore $EC(A_1) = JC(A_1) = S(link_1) = link_1.tc \wedge EC(receive) = link_1.tc \wedge true = link_1.tc = n > 100$

2. In an `if` activity, the branch conditions are evaluated from left to right and the first branch where a condition that evaluates to $true$ is taken. If no condition evaluates to $true$ and no else branch exists, the `if` activity immediately completes. Therefore, for an `if` with n conditional branches with the corresponding conditions $C_1, ..., C_n$ and an else-branch, the execution condition for each branch is constructed as follows:

$$\begin{aligned} \forall i \in [1..n] \ : \ EC(branch_i) \ &= \neg C_1 \wedge ... \wedge \neg C_{i-1} \wedge C_i \\ EC(else) \ &= \ \neg C_1 \wedge ... \wedge \neg C_n \end{aligned}$$

Note that this modelling ensures that the execution condition of the branch j cannot evaluate to $true$, although a branch i exists with $i < j$ and $EC(branch_i) = true$.

As an example, consider the `if` in Figure 1. It contains two branches with the conditions $cond_1 = s > 2.000$ and $cond_2 = s > 1.000$. Thus, the execution conditions of activities A_3 and A_4 become: $EC(A_3) = cond_1$ and $EC(A_4) = \neg cond_1 \wedge cond_2$.

3. In a `pick` activity, the first received message or the timeout event decides, which branch is taken. Without considering any interacting partner, the branch choice is non-deterministic. To model this, each of the branches gets a Boolean attribute *fired*. Assuming that this attribute can only be set to $true$ if and only if the corresponding branch is chosen, the execution condition of each branch in a `pick` P with $B_1, ..., B_n$ branches is modelled as follows:

$$\forall i \in [1..n] \ : \ EC(B_i) \ = \ B_i.fired$$

To model the property that only one of the branches can actually be chosen, the following constraints are used:

$$\begin{gathered} \forall i \in [1..n] \ : \ B_i.fired \rightarrow \forall j \in [1..n](j \neq i \rightarrow \neg B_j.fired) \\ \bigvee_{i \in [1..n]} B_i.fired \end{gathered}$$

A more precise branch-choice semantic has to consider the partner process. Because of the space limitations, we don't consider it here. The complete `pick` semantic modelling is described in [Mon08].

In addition, each structured activity influences its children: if a structured activity is skipped, then all of its children are also skipped. An activity will only be skipped if its execution condition evaluates to false. That means the execution condition of an activity depends on the execution conditions of its parent: if a structured activity P with the execution condition C contains children activities A_1,..., A_n, then the execution conditions of each child A_i takes the form $C_i \wedge C$. Here C_i denotes the combination of other conditions that have impact upon the execution of A_i and is derived as presented above.

In the example shown in Figure 1 the execution conditions of all structured activities, namely `if` and `flow`, are equal $true$. Therefore all derived execution conditions remain unchanged.

Currently the loops are unfolded and thus can be mapped to a set of the `if`-constructs. We investigate the loop invariants to improve this technique. The extension of the proposed approach with `scopes`, `fault-` and `compensation` handlers is also addressed in ongoing work.

3 Analysing BPEL Processes

There are several possible executions of the same BPEL process. Executions vary in the executed activities and in their execution order. Input to the process and variable relations determine which activities are executed and the synchronisation dependencies between activities determine their execution order. The relations between variables are defined in assign activities. If an assign activity is executed, then the relations defined in this activity become valid. Therefore, the execution condition of an assign activity is the enabling condition for the relations defined in this activity. To be able to evaluate an execution condition, we need to know which relations are valid at the point of evaluation time. For this purpose, we need to know which values each variable can take at the point of evaluation. This depends on the activities that may have written to this variable. Such activities are called *possible writers* and can be determined for each variable access using the *Concurrent Static Single Assignment (CSSA)* representation of a BPEL process. In the following, we begin by describing the CSSA form of BPEL in Section 3.1. In Section 3.2 we show how the synchronisation dependencies captured in the CSSA form are modelled using logical assertions. Finally, Section 3.3 presents how the relations between variables are modelled.

3.1 CSSA Representation of BPEL Processes

The *Static Single Assignment (SSA)* form is an intermediate representation that is used to facilitate program analysis and optimisation [C+91]. The SSA form can be characterised through two properties. First, each reference to a name corresponds to the value produced at precisely one definition point giving the single assignment property. The single assignment property is achieved by giving a unique index to each occurrence of the original variable on the left side of an assignment (when it is reassigned). Second, it identifies the points in

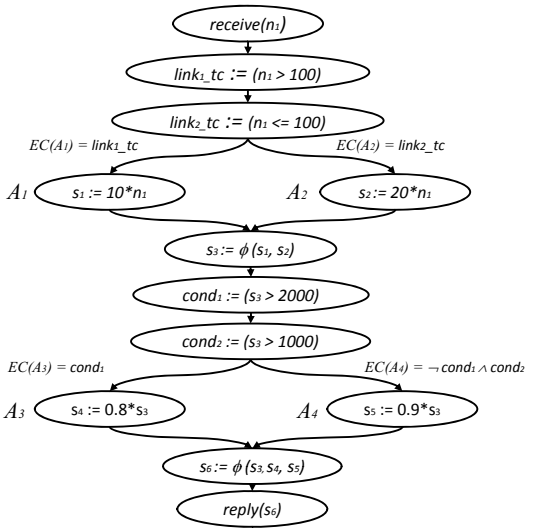

Figure 2: Simplified CSSA representation of the price calculation process

the computation where values from different control-flow paths merge. At a merge point, several different SSA names, corresponding to different definitions of the same original name, may flow together. To ensure the single-assignment property, the construction inserts a new definition at the merge point; its right hand side is a pseudo-function called a ϕ-function that represents the merge of multiple SSA names. As parameters the ϕ-function contains all variables written by possible writers. Due to the uniqueness of variable names, there is no need to distinguish between variables and activities. Thus, we use the term "possible writers" also for the variables, which can be uniquely mapped to the corresponding activity.

While SSA form is suitable for the representation of sequential program execution, it cannot deal with the parallel constructs. To analyse the parallel execution, an extension to the SSA form, called *Concurrent Static Single Assignment (CSSA)* is used [LMP97]. The idea of the CSSA form is that it summarises the interleaving information for conflicting variables in an explicitly parallel program through the use of π-functions. The values of all conflicting variables are well defined by the π-function at the point where the π-function is placed and are represented via parameters of this function. Like the SSA form, the CSSA form has the property that all uses of a variable are reached by exactly one assignment to the variable.

The approach described in [M$^+$07] shows how to obtain a CSSA representation for a BPEL process. Figure 2 shows the simplified CSSA representation of the price calculation process from Figure 1. For readability, the nodes representing implicit join conditions, which in our case are equal to the status of the incoming link, are skipped.

The CSSA representation captures all accesses to the process variables. The link transition conditions, the activity join conditions and the `if` branch conditions are each represented as single nodes. Each node defines a unique variable that represents the

corresponding condition. These variables are used for modelling an activity's execution conditions.

3.2 Modelling Synchronisation Dependencies

The synchronisation dependencies are captured within the CSSA representation. An activity B has a synchronisation dependency on activity A if there exists a path from A to B in the CSSA graph. We assume that each assign activity writes only one variable, and thus that dependencies between activities can be considered as dependencies between corresponding written variables.

For each assign activity A let W_A denote the variable written in A. We define a *Direct Dependency Set* $D(W_A)$ as follows: $D(W_A) = \{W_{A'} \mid$ there is a path from A' to A which does not contain any other assign activity$\}$

The synchronisation dependencies define the irreflexive partial order on activities execution. To model this partial order, and therefore the constraints on possible executions, each variable W_A gets an $order$ attribute. This attribute is of type Integer. Our goal is to specify the constraints on how these attributes can be assigned and an SMT solver assigns the concrete values.

Let $\mathcal{A}$ denote the set of all assign activities. We constrain all possible executions of the assign activities to those that are allowed by the specified synchronisation dependencies as follows:

$$\forall A \in \mathcal{A}, \forall W_{A'} \in D(W_A) : \; W_A.order > W_{A'}.order$$

Note that parallel activities do not have any synchronisation dependencies. Therefore, the variables written in such activities do not get mutual constraints. This corresponds to the non-determinism within parallel constructs.

3.3 Modelling Relations Between Variables

The relations between variables are defined by assign activities. The evaluation of the execution condition of an assign activity decides whether this relation is valid. To model these relations each variable gets an ec and a val attributes. The Boolean ec attribute defines the execution condition of the activity, in which this variable is written. The val attribute denotes the value of this variable. Note that the actual values are not necessarily known. For example, for the assign activity A_1: $s_1 := 10 * n_1$ we specify a constraint $s_1.val = 10 * n_1.val$. Even if the actual value of n_1 is not known, the assertion captures the dependency between the values of s_1 and n_1.

An assign activity A will only be executed if its execution conditions $EC(A)$ evaluates to true. Let W_A denote the variable written in A and $f(x_1, ..., x_n)$ denote the right side of A. In this work $f(x_1, ..., x_n)$ can only be a ϕ-, π or a linear arithmetic function. The

relation between variables defined by A is modelled as follows:

$$W_A.ec = EC(A)$$
$$W_A.ec \rightarrow W_A = f(x_1, ..., x_n)$$

According to these rules, we specify the following assertions for A_3 and A_4:

$$cond_1 = s_3 > 2000 \qquad cond_2 = s_3 > 2000$$
$$s_4.ec = cond_1 \qquad s_4.ec \rightarrow s_4 = 0.8 * s_3$$
$$s_5.ec = \neg cond_1 \wedge cond_2 \qquad s_5.ec \rightarrow s_5 = 0.9 * s_3$$

In case $f(x_1, ..., x_n)$ is a linear arithmetic expression, it is mapped one to one, as the example for the activities A_3 and A_4 above shows. To model the assigns with the ϕ- and π- functions on the right side, we need to specify their selection semantics. The *selection semantics* defines the rules for the selection of the effective writer. In each single execution of a process each variable has only one effective writer, namely the one that wrote the variable which is used in the current execution. For example, consider the `if` construct in Figure 1. There are three possible executions for this `if`: the left branch is taken, the right branch is taken or the `if` is skipped. For each of these executions the value of s_6 is clearly defined: in the case of the left branch it becomes s_4, in the case of the right branch it becomes s_5 and in the skipped case it becomes s_3. This describes the selection semantic of the $\phi(s_3, s_4, s_5)$-function for our example.

The ϕ-function is synchronised on the possible writers listed as parameters of the ϕ-function. Therefore, the last writer is the effective writer. Thus, the selection semantic of a ϕ-function is modelled as follows: If $x_i = \phi(x_{i_1}, ..., x_{i_n})$, then we define the following constraints on the value of x_i, where x_{i_k} denotes the effective writer:

$$\bigvee_{k \in [1,n]} \left((x_i.val = x_{i_k}.val) \wedge x_{i_k}.ec \wedge \bigwedge_{l \in [1,n], l \neq k} (x_{i_l}.ec \rightarrow (x_{i_k}.order > x_{i_l}.order)) \right)$$

For our example, the ϕ-node $s_6 = \phi(s_3, s_4, s_5)$ is modelled by the following assertion:

$$\begin{aligned}
&(s_6.val = s_3.val) \wedge s_3.ec \wedge (s_4.ec \rightarrow (s_3.order > s_4.order)) \\
&\qquad \wedge (s_5.ec \rightarrow (s_3.order > s_5.order)) \\
\vee\ &(s_6.val = s_4.val) \wedge s_4.ec \wedge (s_3.ec \rightarrow (s_4.order > s_3.order)) \\
&\qquad \wedge (s_5.ec \rightarrow (s_4.order > s_5.order)) \\
\vee\ &(s_6.val = s_5.val) \wedge s_5.ec \wedge (s_3.ec \rightarrow (s_5.order > s_3.order)) \\
&\qquad \wedge (s_4.ec \rightarrow (s_5.order > s_4.order))
\end{aligned}$$

The assertions specified for A_3 and A_4 together with the above assertion for the ϕ-function model the relations between the variables s_3, s_4, s_5, s_6, $cond_1$, $cond_2$. For example, the value assignments $s_3 = 1000$; $cond_1 = false$; $cond_2 = false$; $s_4 = any$; $s_5 =$

any; $s_6 = s_3 = 1000$ satisfy these assertions, while the assignment $s_3 = 1000$; $cond_1 = false$; $cond_2 = false$; $s_4 = any$; $s_5 = any$; $s_6 = s_3 = 900$ does not.

While a ϕ-function chooses an effective writer after all possible writers are executed, the assertions for the π-function should consider the possibility that some of the possible writers defined in the π-function can actually be executed after the assign activity that uses the value of the π-function. In addition, because the possible writers of the π-function are not necessarily synchronised, we have to explicitly model the property that a variable has to be written before it can be used as the value of the π-function. Collectively, we model the selection semantic of a π-function as follows:

If $x_i = \pi(x_{i_1}, ..., x_{i_n})$, then we define on the value of x_i the following constraints:

$$\bigvee_{k \in [1,n]} \Bigg((x_i.val = x_{i_k}.val) \wedge x_{i_k}.ec \wedge (x_i.order > x_{i_k}.order) \wedge \bigwedge_{l \in [1,n], l \neq k} (x_{i_l}.ec \rightarrow ((x_{i_k}.order > x_{i_l}.order) \vee (x_{i_l}.order > x_i.order))) \Bigg)$$

Compared to the ϕ-function, we have two new clauses. $x_i.order > x_{i_k}.order$ models the fact that x_{i_k} has to be written before x_i can read it and $x_{i_l}.ec \rightarrow ((x_{i_k}.order > x_{i_l}.order) \vee (x_{i_l}.order > x_i.order))$ states that if any other possible writer is written ($x_{i_l}.ec = true$), then it is either written before x_{i_k} or after x_i. Otherwise x_i would read the value of x_{i_l}.

4 Business Constraints Verification

In the previous section we showed how the logical assertions could be used to capture the relationships between BPEL process variables. These assertions form the basis for the business constraints verification. To verify a business constraint, its negation is modelled as an assertion and added to the evaluation basis. If the obtained combination of assertions cannot be satisfied, then the business constraint itself is fulfilled. Otherwise an assignment to the process variables violating this constraint will be found.

Let C denote the conjunction of the assertions modelling the verification basis. Note that C is fulfilled for the modelled process. For a given business constraint let B denote the corresponding logical assertion. For the business constraint to be always fulfilled for the modelled process the formula $C \rightarrow B$ has to be valid. To prove this, we verify the satisfiability of its negation:

$$\neg(C \rightarrow B) = C \wedge \neg B$$

If it cannot be satisfied, then $C \rightarrow B$ is valid and therefore the business constraint B is fulfilled for our process model.

As an example we show how the business constraints defined in Section 1 can be verified for the price calculation process. We assume that the basis is already modelled as shown in

Section 3. So far this basis is satisfiable with all assignments to the variables, which are possible in the real process execution.

Assume we want to verify the following business constraint: "A customer who ordered less than 200 items should not receive a discount of more than 10%". The negation of this rule is represented with the following assertion:

$$(n_1.val < 200) \wedge s_4.ec$$

Here s_4 is the variable written in the activity A_3, which calculates a 20% discount. This assertion assumes that there is a possible configuration of the variables that satisfies the fact that the customer orders less than 200 items and gets 20% discount. Alternatively it can be expressed with the following assertion:

$$(n_1.val < 200) \wedge (s_6.val < 0.9 * s_3.val)$$

If one of the above assertions is added to the modelled verification basis, it becomes unsatisfiable. In other words, the business constraint is fulfilled. This is due to the fact that the dependency between n and s is captured within the basis assertions. Therefore if $100 < n_1 < 200 \Rightarrow link_1\_tc = true \Rightarrow s_1 = 10 * n_1 < 2000 \Rightarrow s_3 = s_1 \Rightarrow cond_1 = false \Rightarrow$ the customer will not get 20% discount. If $n_1 \leq 100 \Rightarrow link_2\_tc = true \Rightarrow s_2 = 20 * n_1 < 2000 \Rightarrow s_3 = s_2 \Rightarrow cond_1 = false \Rightarrow$ customer will not get 20% discount.

Analogous to the above, for the property "If the customer ordered 50 items or more, then they should receive a discount" the corresponding assertion, which corresponds to the negation of this property, is:

$$\neg((n_1.val \geq 50) \rightarrow (s_4.ec \vee s_5.ec))$$

This assumes that a customer who ordered 50 or more items did not receive a discount. Alternatively, this rule can be modelled as follows:

$$\neg((n_1.val \geq 50) \rightarrow (s_6.val < s_3.val))$$

In this case, Yices finds variable assignments that satisfy the modelled context. This model contains $n_1 = 50$, which means that when the customer orders 50 items, they do not receive any discount[2].

5 Related Work

An overview of existing BPEL formalizations and verification approaches is provided in [BK06]. We present a summary of the presented approaches here.

[2]The input for Yices is available at `http://www.iaas.uni-stuttgart.de/forschung/pricecalculation.ys`.

Petri net approaches abstract from data-flow [MM06, Loh07]. For example, the decision of which `if` branch is taken is made non-deterministically. Thus, in the example from Figure 1 the execution path $receive$, A_2, A_3, $reply$ becomes possible, which can never happen and thus leads to wrong verification results. The approach of [YTYL05] transforms BPEL to Coloured Petri nets. Here, each type of message is transformed to a token with a different colour. However, the mapping does not consider relations between the conditions and variables.

The Promela approach transforms a BPEL process into Promela and verifies it with the SPIN model checker [FBS04, Nak05, FFK05]. SPIN itself cannot handle large data domains. SPIN works with explicit states and therefore has to check all possible values an integer variable n can take, which is infinite [Hol04]. Even if the value of n is bounded by the maximum integer value, the number of states explodes. For example, if x is bound to $[r_1 \dots r_2$ and y to $[s_1 \dots s_2$, then SPIN has to consider $(r_2 - r_1 + 1) * (s_2 - s_1 + 1)$ states. The approach presented in [BGS07] is similar to the Promela approaches, but uses Bogor [RDH03] to do the actual model checking. As the Promela approach, it cannot handle the large data domains.

An approach based on abstract state machines (ASM) is presented in [FR05]. While the mapping covers scopes, it does not consider the relations between conditions and variables. Approaches based on the π-calculus are presented in [WDW07, Fad04, LM07]. As with the ASM approaches, these do not consider the relations between conditions and variables.

The approach presented in [PA08] transforms a BPEL process into a Java program using B2J [Ecl08]. The Java model checker Java PathFinder (JPF, [VHB+03]) is then used with the transformed program. The model checker uses explicit states for each variables combination and therefore cannot handle large or unbound data domains.

When it comes to the determination of data-flow in BPEL processes, the control-flow has to be analyzed. Current work on data-flow analysis are presented in [M+07], [KKL08] and [ZZK07]. The approach presented in [M+07] is based on CSSA, the approach of [KKL08] is based on abstract interpretation, and the approach of [ZZK07] is based on automatons. All of these approaches determine a set of possible writers for each use of a variable. However, all of them do not consider variable relations and the selection semantics. Thus, all of them return $\{A_1, A_2\}$ as possible writers for A_3, which is an over-approximation. This over-approximation can be improved if the execution conditions for each activity were considered.

We showed in [Mon08] how the approach can be used to model and to verify service communication. A proof of concept has been provided using IBM WebSphere as implementation platform.

6 Conclusion and Outlook

We analysed the relations between variables and showed their influence on the data-flow. The presented verification algorithm uses the results of this analysis and enables verification of business constraints. The BPEL process execution semantics, the variable relations and

the business constraints were modelled using logical assertions. These assertions were verified using the SMT solver Yices, that is extensible with different theories. In this work we presented how the linear arithmetic theory can be used to enable business rules verification. Our approach is the first one which goes beyond simple control flow analysis and considers the dependency between control flow and data flow. That is not possible with the other approaches.

The mapping to logical assertions presented in this work excluded BPEL scopes, which is a part of our ongoing work. Furthermore, we investigate the other possibilities to handle the loop constructs, e.g. using loop invariants. We also plan to use the results of this work to analyse interacting processes and choreographies expressed in BPEL4Chor [D+07]. This should enable the analysis and combination of the variables relationships between different partners as shown in [Mon08].

Acknowledgments The work published in this article is partially funded by the MASTER project under the EU 7th Framework Programme (contract no. FP7-216917). Oliver Kopp is funded by the German Ministry of Education and Research (project Tools4BPEL, project number 01ISE08B).

References

[B+06] Bernhard Beckert et al. Intelligent Systems and Formal Methods in Software Engineering. *IEEE Intelligent Systems*, 21(6):71–81, 2006.

[BGS07] Domenico Bianculli, Carlo Ghezzi, and Paola Spoletini. A Model Checking Approach to Verify BPEL4WS Workflows. In *IEEE International Conference on Service-Oriented Computing and Applications (SOCA '07)*, pages 13–20. IEEE computer society, 2007.

[BK06] Franck van Breugel and Maria Koshkina. Models and Verification of BPEL. `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf`, 2006.

[C+91] Ron Cytron et al. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.

[C+03] Francisco Curbera et al. Exception Handling in the BPEL4WS Language. In *Conference on Business Process Management*, pages 276–290. Springer, 2003.

[D+07] Gero Decker et al. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *IEEE International Conference on Web Services*. IEEE Computer Society, 2007.

[DdM08] Bruno Dutertre and Leonardo de Moura. The YICES SMT Solver, 2008. Available at `http://yices.csl.sri.com/`.

[Ecl08] Eclipse Foundation. BPEL to Java (B2J) Subproject, 2008. `http://www.eclipse.org/stp/b2j/`.

[Fad04] M. Fadlisyah. Using the π-Calculus for Modeling and Verifying Processes on Web Services. Master's thesis, Insitute for Theoretical Computer Science, Dresden University of Technology, 2004.

[FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. Model checking XML manipulating software. In *IEEE Int. Symp. on Software Testing and Analysis*. ACM, 2004.

[FFK05] Jesús Arias Fisteus, Luis Sánchez Fernández, and Carlos Delgado Kloos. Applying model checking to BPEL4WS business collaborations. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 826–830. ACM, 2005.

[FR05] D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative Control Flow. In *12th International Workshop on Abstract State Machines*, pages 131–151, March 2005.

[Hol04] Gerard J. Holzmann. *SPIN Model Checker, The: Primer and Reference Manual*. Addison-Wesley Professional, 2004.

[KKL08] Oliver Kopp, Rania Khalaf, and Frank Leymann. Deriving Explicit Data Links in WS-BPEL Processes. In *IEEE International Conference on Services Computing*. IEEE Computer Society Press, 2008.

[LM07] Roberto Lucchia and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, January 2007.

[LMP97] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 1997.

[Loh07] Niels Lohmann. A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In *International Workshop on Web Services and Formal Methods*. Springer, 2007.

[M+07] Simon Moser et al. Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis. In *IEEE International Conference on Services Computing*, July 2007.

[MM06] Axel Martens and Simon Moser. Diagnosing SCA Components UsingWombat. In *Conference on Business Process Management*. Springer, 2006.

[Mon08] Ganna Monakova. Ontology Based Partner Service Discovery Using a First-Order Logic Representation for BPEL Process Models. Diploma thesis, University of Stuttgart, Institute of Architecture of Application Systems, 2008.

[Nak05] Shin Nakajima. Lightweight formal analysis of Web service flows. *Progress in Informatics*, 1:57–76, November 2005.

[ND79] G. Nelson and Oppen D. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[OAS07] OASIS. *Web Services Business Process Execution Language Version 2.0*, 2007.

[PA08] P. Parizek and J. Adamek. Checking Session-Oriented Interactions between Web Services. In *Proceedings of 34th EUROMICRO SEAA conference*, pages 3–10. IEEE Computer Society, 2008.

[RDH03] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *9th European software engineering conference (ESEC/SIGSOFT FSE)*, pages 267–276. ACM, 2003.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[WDW07] Matthias Weidlich, Gero Decker, and Mathias Weske. Efficient Analysis of BPEL 2.0 Processes using pi-Calculus. In *Proceedings of the IEEE Asia-Pacific Services Computing Conference (APSCC)*. IEEE Computer Society, 2007.

[YTYL05] YanPing Yang, QingPing Tan, JinShan Yu, and Feng Liu. Transformation BPEL to CP-Nets for Verifying Web Services Composition. In *International Conference on Next Generation Web Services Practices (NWeSP)*, pages 137–142. IEEE Computer Society, 2005.

[ZZK07] Yongyan Zheng, Jiong Zhou, and Paul Krause. Analysis of BPEL Data Dependencies. In *Proceedings of the 33rd EUROMICRO SEAA conference*, pages 351–358. IEEE Computer Society, 2007.

All links were last followed on October 20, 2008.