Deriving Explicit Data Links in WS-BPEL Processes

Oliver Kopp[1], Rania Khalaf[2], Frank Leymann[1]

[1]Institute of Architecture of Application Systems, University of Stuttgart, Germany
{kopp,leymann}@iaas.uni-stuttgart.de

[2]IBM TJ Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
rkhalaf@us.ibm.com

Deriving Explicit Data Links in WS-BPEL Processes

Oliver Kopp[1], Rania Khalaf[2], Frank Leymann[1]
[1]Institute of Architecture of Application Systems, University of Stuttgart, Germany
{kopp,leymann}@iaas.uni-stuttgart.de
[2]IBM TJ Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
rkhalaf@us.ibm.com

Abstract

WS-BPEL is a standard language to model business processes. Control flow is modeled explicitly using links. Data is passed via shared variables and there is no notion of explicit data links. However, explicit data links are an important means to reason about business process models. We present an algorithm to derive explicit data links in WS-BPEL processes. By considering dead path elimination as defined in WS-BPEL, we reduce the number of derived data links when compared to existing approaches that ignore dead path elimination.

1. Introduction

The Web Service Business Process Execution Language (BPEL for short) is a workflow language geared towards Service Oriented Computing. BPEL provides basic workflow capabilities, such as the ability to define the control flow between a set of activities via explicit links, as well as advanced features such as recovery, fault handling and event handling. The area of workflow often requires both, understanding the control flow as well as the data flow between a set of activities. This aids the design of business processes as well as their analysis and reengineering. BPEL, however, has no explicit construct for modeling data flow, but uses variables shared between activities for that purpose: Activities simply read from and write to these variables. In order to derive explicit data links between the activities of a BPEL process, one must therefore perform a data-flow analysis on that process.

To address this problem, we present an algorithm that statically determines data links. Mainstream data-flow analysis techniques are presented in [1, 13, 14]. However, these techniques cannot be directly applied to a BPEL process, since BPEL supports both, parallelism and dead path elimination (DPE) [15]. DPE is a technique used in BPEL (as well as many workflow system implementations) to propagate the disablement of activities that can no longer be executed. While algorithms are presented that determine data dependencies in BPEL [5, 12], these algorithms typically produce too many data links. The algorithm presented in this paper determines explicit data links and deals with DPE, thus reducing the number of data links.

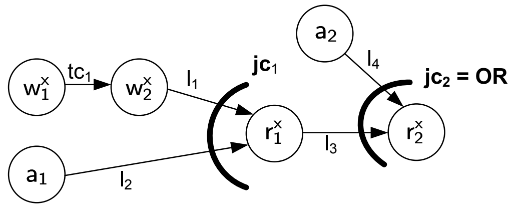
Explicit data links are used in [12] to construct a more precise formal model of the given BPEL process. The work of [5] uses data links to determine uninitialized variables. We create a data-flow analysis algorithm for BPEL to enable splitting BPEL processes based on business needs: A user assigns the activities of a business process to different partners and the result is a set of BPEL processes, one for each partner, such that the operational semantics of the original process is maintained. The splitting of BPEL processes is introduced in [8]. In that work data dependencies are modeled explicitly in BPEL by using BPEL-D, which is an extension of BPEL [6].

We present in [7] an algorithm to split standard BPEL processes. One of the inputs to that algorithm is the result of data-flow analysis on the BPEL process being split. A data link between two activities in different fragments is roughly translated into a message sent from one fragment to another. Therefore, it is important to return as few data links as possible in order to minimize the communication overhead between the partners. This work relates to [7] in that it provides one specific data-flow analysis algorithm determining the data links in a BPEL process and providing a small number of data links due to its treatment of DPE.

The remainder of the paper is structured as follows: Section 2 presents the challenges of DPE for data-flow analysis and section 3 presents an algorithm addressing the challenges. Section 4 presents the state of the art in the field of data-flow analysis on BPEL processes and section 5 draws a conclusion and presents future work.

2. Challenges of DPE

In order to address data-flow analysis in BPEL, we present a short summary of the behavior of links and activities focusing on processes where `suppressJoinFailure` is set to `yes`, thus enabling dead path elimination. Each BPEL activity may have incoming and/or outgoing links, where each link is associated with a transition condition. If no explicit transition condition is associated, the default transition condition is `true`. A BPEL join condition is a Boolean function over the status of the incoming links of an activity. Once every incoming link has fired, the join condition is evaluated. If the join condition evaluates to `true`, the activity is executed, and if executed successfully, the transition conditions of the outgoing links are evaluated. If the join condition evaluates to `false`, the activity is not executed and the status of its outgoing links is set to `false`. Regardless of whether the activity is executed or not, the outgoing links are fired and the target activities visited. This technique is called "dead path elimination" (DPE), which is explained in detail in [4]. Activities and links not being executed due to DPE are called *dead*. It should be noted that the existence of multiple incoming links associated with an activity always represents a synchronizing join.



w^x	activity writing to variable x
r^x	activity reading from variable x
a	activity not reading or writing
jc	join condition
tc	link with explicit transition condition
l	link with default transition condition `true`

Figure 1. Process with join conditions

Figure 1 presents activities in a BPEL flow. A flow is a compound activity that contains activities and control dependencies between them (i.e., a directed acyclic graph). The contained activities are represented as annotated nodes. The annotation shows whether the activity is reading from the variable x, writing to the variable x, or neither reading from nor writing to it.

Consider the join condition jc_1. Whether w_1^x's write reaches r_1^x depends on the value of join condition jc_1. Suppose tc_1 evaluates to `false`. Then w_2^x is marked as dead and the status of l_1 is set to `false`. If jc_1 is a logical AND over the status of all incoming links, r_1^x will be dead and will therefore never read the data written by w_1^x. However, w_1^x can still be a valid writer for a subsequent read, after r_1^x: Consider that the status of l_4 evaluates to `true`, leading to r_2^x being executed even though w_2^x and r_1^x are dead. Since w_1^x has been executed, a data link from w_1^x to r_2^x must be created.

Now suppose jc_1 is instead a logical OR[1]. The evaluation of jc_1 will always return `true`, regardless of tc_1, since l_2 has no explicit transition condition assigned and a_1 is never dead. If tc_1 evaluates to `false`, w_2^x is dead. Thus, r_1^x reads the value written by w_1^x and not the value written by w_2^x.

All in all, explicit data links are dependent on the join conditions: If jc_1 is a logical OR, both w_1^x and w_2^x are possible writers for r_1^x. If jc_1 is a logical AND, w_2^x is the only possible writer for r_1^x.

3. The Algorithm

In the following, we present an algorithm which determines data links in BPEL processes and respects dead-path elimination behavior. The algorithm is based on the ideas presented in [14], where the control flow is abstractly interpreted.

To stay close to existing syntax formalizations of BPEL, our notation is based on the one presented in [16], where a detailed explanation of the notation can be found. The used notation is presented in table 1. The central idea of the formalization is to connect nested XML elements using a relation $HR \subseteq \mathscr{A} \times \mathscr{A}$. If c is an XML child of a `flow` activity f, then $(f,c) \in \mathsf{HR}$. Control links specified in a flow activity are explicitly modeled by the set of control links $\mathscr{L}$. The control flow is modeled by the control link relation $\mathsf{LR} \subset \mathscr{A} \times \mathscr{L} \times \mathscr{A}$, where $(p,l,s) \in \mathsf{LR}$ denotes that activity p is connected to activity s by the control link l.

3.1. Handling Complex Types

Variables in BPEL processes are accessed using queries. Let $Q_{\mathscr{V}}$ denote the set of all queries for accessing locations in variables. In the context of XPath [17], such a query is a location path. If the whole variable is accessed, the query is empty (ε). We define $E^{Q_{\mathscr{V}}} \subseteq \mathscr{V} \times Q_{\mathscr{V}}$ to denote all tuples of variables and queries on each variable in the given process model. In the subsequent sections, each element in $E^{Q_{\mathscr{V}}}$ is called *variable element* to ease reading. The set $E_w^{Q_{\mathscr{V}}} \subseteq E^{Q_{\mathscr{V}}}$ is the set of all variable elements, which are written by

[1] We use the term "jc is a logical OR" as shortcut for "jc is a Boolean OR function over the status of *all* incoming links". Similar for "jc is a logical AND".

Table 1. Notations used

Notation	Meaning
$\mathscr{A}_{basic}$	The set of all basic activities
$\mathsf{process}$	The `process` element
$\mathscr{A}_{flow}$	The set of all flow activities
$\mathscr{A}$	The set of all activities; $\mathscr{A} = \mathscr{A}_{basic} \cup \{\mathsf{process}\} \cup \mathscr{A}_{flow}$
$\mathscr{C}$	Set of all Boolean conditions
$\pi_i(t)$	Returns the projection to the i^{th} component of a tuple t
$\mathscr{L}$	All control links in `flow` activities
$\mathsf{LR} \subset \mathscr{A} \times \mathscr{L} \times \mathscr{A}$	The control link relation
$\wp(S)$	Denotes the power set of a set S
$\mathscr{L}_{in} : \mathscr{A} \to \wp(\mathscr{L})$	Returns the set of all incoming links of an activity
$\mathscr{L}_{out} : \mathscr{A} \to \wp(\mathscr{L})$	Returns the set of all outgoing links of an activity
$\mathsf{jc} : \mathscr{A} \to \mathscr{C} \cup \{\perp\}$	Returns the join condition of the given activity. "$\perp$" denotes an undefined join condition defaulting to the logical OR of all incoming links.
$\mathsf{HR} \subseteq \mathscr{A} \times \mathscr{A}$	The hierarchy relation denoting the nesting of activities. Let $h = (p, a) \in \mathsf{HR}$. Then p is a parent of a. The execution order of activities nested in a flow activity is specified by the control link relation LR.
$\mathsf{children} : \mathscr{A} \to \wp(\mathscr{A})$	Returns the set of all children of the given activity with respect to HR
$\mathsf{descendants} : \mathscr{A} \to \wp(\mathscr{A})$	Returns the set of all transitive children of the given activity with respect to HR
$\mathscr{V}$	Set of all variables including the variable implictly declared at fault handlers
$Q_{\mathscr{V}}$	Set of all queries (i.e., XPath location paths) for accessing locations in variables. $Q_{\mathscr{V}}$ includes 'ε' to denote "empty query", i.e., the whole variable is accessed.
$E^{Q_{\mathscr{V}}} \subseteq \mathscr{V} \times Q_{\mathscr{V}}$	The set of tuples of a variable v and a (valid) query q on v
$E_w^{Q_{\mathscr{V}}} \subseteq E^{Q_{\mathscr{V}}}$	The set of variable elements written by activities
$w : \mathscr{A} \times E^{Q_{\mathscr{V}}} \to \mathbb{B}$	Returns `true` iff the given activity completely changes the given variable element and not only parts of it
$r : \mathscr{A} \cup \mathscr{L} \times E^{Q_{\mathscr{V}}} \to \mathbb{B}$	Returns `true` iff the given activity or link completely reads the given variable element

activities.

Let `$var` be a variable of a complex type and thus being of the form presented in figure 2. Assume now three writers $\mathsf{w}_1^{\mathsf{v}}$, $\mathsf{w}_2^{\mathsf{v/c}}$, $\mathsf{w}_3^{\mathsf{v/c}}$ in a sequence with following writes: $\mathsf{w}_1^{\mathsf{v}}$: `$var`, $\mathsf{w}_2^{\mathsf{v/c}}$: `$var/car`, $\mathsf{w}_3^{\mathsf{v/c}}$: `$var/car`. $\mathsf{w}_3^{\mathsf{v/c}}$ overwrites the data written by $\mathsf{w}_2^{\mathsf{v/c}}$, but it does not overwrite all data written by $\mathsf{w}_1^{\mathsf{v}}$. Thus, $\mathsf{w}_1^{\mathsf{v}}$ is still a writer to take into account for a subsequent read on `$var`, where the data written by $\mathsf{w}_1^{\mathsf{v}}$ and $\mathsf{w}_3^{\mathsf{v/c}}$ has to be merged.

A writer has to be removed from the set of possible writers if its written data is overwritten by a subsequent write (cf. section 3.4). Therefore, every writer writing to a variable element v_e also has to be considered as a writer to elements being subelements of v_e. To formally define

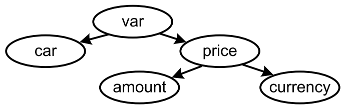


Figure 2. Variable of a complex type

the subelement-relation, we can interpret the structure of each variable v as a lattice. Assume the comparison operator of the lattice to be "$\sqsupset$". Then, $n \sqsupset m$ returns `true` iff n is a parent of m in the structure of the variable v, where n and m are parts of the variable. Thus, $\sqsupset$ forms the subelement-relation.

XPath expressions may reference more than one location in the structure of a variable. The presented approach for handling elements in the variable structure works for single elements only. Therefore, we restrict XPath queries not to contain any variable reference and to return exactly one element in a variable.

Assume $\mathsf{poss}_\circ : (\mathscr{A} \cup \mathscr{L}) \times E^{Q_{\mathscr{V}}} \to \wp(\mathscr{A})$ being the function returning the set of possible writers for a given activity or link and variable element during program execution (cf. section 3.2). An activity is contained in $\mathsf{poss}_\circ$ if it completely writes to v_e. As shown above, a read of an activity a on a variable element v_e also has to read all data written to children of v_e. Therefore, the function $\mathscr{W}_{\between} : \mathscr{A} \cup \mathscr{L} \times E^{Q_{\mathscr{V}}} \to \wp(\mathscr{A})$ returning the set of possible writers, including partial writers, is defined as follows:

$$\mathscr{W}_{\not\parallel} : (x, v_e) \mapsto \{a' \mid a' \in \mathsf{poss}_\circ(a, v'_e), v'_e \sqsubseteq v, r(x, v_e)\}$$

$r : \mathscr{A} \cup \mathscr{L} \times E^{Q_\mathscr{V}} \to \mathbb{B}$ returns `true` iff the given activity a directly reads the given variable element v_e.

3.2. Static Analysis

The control flow of the BPEL process definition is abstractly interpreted. Since DPE may only be active within a flow activity, the algorithm handles BPEL processes including a single `flow` activity as the only structured activity. We assume that the Bernstein Criterion [11] holds for the analyzed BPEL process. The Bernstein Criterion states that there may be no writes happening in parallel to reads on the same variable.

The idea of the static analysis is to distinguish between three states of a writing activity: possible, disabled and invalid. A writing activity w is a possible writer at an activity a if the data written by w can reach a. For example, w_2^x is a possible writer for r_1^x in the BPEL process presented in figure 1. In general, w is a disabled writer if it is overwritten by a subsequent writer. If the process execution reaches the activity w_2^x, w_1^x gets overwritten by w_2^x and thus w_1^x is disabled. If the join condition jc_1 is an AND, w_1^x will also remain disabled at r_1^x: If tc_1 evaluates to `false`, w_2^x is dead. Thus, the status of l_1 is `false` and jc_1 subsequently evaluates to `false`. Thus, r_1^x is dead and will not read the value written by w_1^x. A disabled writer can be a possible writer again, which happens for w_1^x at r_2^x, if jc_1 is an AND. Since l_4 does not contain an explicit transition condition, the status of l_4 is `true` and the join condition at r_2^x evaluates to `true` even if the status of l_3 is `false`. Thus, the data written by w_1^x does not reach r_1^x, but r_2^x. A writer can also be disabled completely and thus become invalid. Assume tc_1 is `true`. Then w_2^x always overwrites the value written by w_1^x and thus w_1^x is never a possible writer at all activities following w_2^x.

To store the state of a writing activity, two lattices and a Boolean value are used: One lattice for the possible writers, another lattice for the disabled writers, and one Boolean value for marking the current activity to be possibly dead due to DPE. The invalid writers are not stored, since they are not needed to determine the possible writers. Each lattice is formed in the same way: Each element of the lattice is a subset of the set of all activities, and the containment relation forms the lattice relation. The current state of the writes to the given variable element is assigned to every link and activity by the function $\mathsf{writes}_\circ$. The function $\mathsf{writes}_\bullet$ returns the current states of the writes after the activity or the link has been interpreted.

$$\mathsf{writes}_\circ : (\mathscr{A} \cup \mathscr{L}) \times E^{Q_\mathscr{V}} \to \wp(\mathscr{A}) \times \wp(\mathscr{A}) \times \mathbb{B}$$
$$\mathsf{writes}_\bullet : (\mathscr{A} \cup \mathscr{L}) \times E^{Q_\mathscr{V}} \to \wp(\mathscr{A}) \times \wp(\mathscr{A}) \times \mathbb{B}$$

To ease reading, the following functions are used to access each tuple element:

$\mathsf{poss}_\circ(x, v_e) := \pi_1(\mathsf{writes}_\circ(x, v_e))$ returns the activities which are possible writes to the variable element v_e at position x in the BPEL process. A position x can be an activity or a link. “poss” stands for “possible”.

$\mathsf{dis}_\circ(x, v_e) := \pi_2(\mathsf{writes}_\circ(x, v_e))$ returns the activities which were overwritten by preceding writes from a path leading from the root to position x. “dis” stands for “disabled”. The writers contained in $\mathsf{dis}_\circ$ are those writers, which may get possible writers at a subsequent OR join as outlined in the example.

$\mathsf{mbd}_\circ(x, v_e) := \pi_3(\mathsf{writes}_\circ(x, v_e))$ returns `true` iff x may be not executed due to DPE on a path from any directly preceding writer (or from the root node if there is no directly preceding writer) to x. “mbd” stands for “may be dead”. $\mathsf{mbd}_\circ(x, v_e)$ is needed to decide if a writer x disables preceding possible writers or if x makes them invalid. A writer w is a directly preceding writer if there is no other writer on the path from w to x.

$\mathsf{poss}_\bullet$, $\mathsf{dis}_\bullet$, and $\mathsf{mbd}_\bullet$ are defined on $\mathsf{writes}_\bullet$ similarly to $\mathsf{poss}_\circ$, $\mathsf{dis}_\circ$, and $\mathsf{mbd}_\circ$ on $\mathsf{writes}_\circ$.

3.3. Depth-first Search

The static analysis executes a depth-first search (DFS) which covers all possible executions. Since each link in the flow graph is associated with a transition condition, the DFS visits the links explicitly. A DFS is started for each variable element $v_e \in E_w^{Q_\mathscr{V}}$ as shown in Algorithm 1. Variable elements can be analyzed independently from each other, since BPEL does not support aliasing of variables. “Aliasing” describes the fact that two variables can point to the same place in memory.

The DFS is implemented in HANDLEACTIVITY and HANDLELINK. HANDLELINK visits the control links and is described in section 3.7. The activities are visited by HANDLEACTIVITY, which is presented in Al-

Algorithm 1 Analysis of a given BPEL process

```
procedure ANALYZEPROCESSMODEL
    Determine $E_w^{Q_\mathscr{V}}$ in the given BPEL process model.
    for all $v_e \in E_w^{Q_\mathscr{V}}$ do
        $\forall a \in \mathscr{A} : \mathsf{visited}(a) \leftarrow$ false
        $\forall l \in \mathscr{L} : \mathsf{visited}(l) \leftarrow$ false
        HANDLEACTVITY(process, $v_e$)
    end for
end procedure
```

Algorithm 2 Handling of an activity

```
procedure HANDLEACTIVITY($a$,$v_e$)
    parentHandled ←
            $a = \mathsf{process} \vee \mathsf{visited}(p),\ (p,a) \in \mathsf{HR}$
    allLinksVisited ←
            $\mathscr{L}_{in}(a) = \emptyset \vee \forall l \in \mathscr{L}_{in}(a) : \mathsf{visited}(l)$
    if parentHandled ∧ allLinksVisited then
        $\mathsf{visited}(a) \leftarrow \texttt{true}$
        $\mathsf{writes}_\circ(a, v_e) \leftarrow \begin{cases} \mathsf{joinLinks}(a, v_e) & |\mathscr{L}_{in}(a)| > 0 \\ \mathsf{writes}_\circ(p, v_e) & \exists p : (p,a) \in \mathsf{HR} \\ (\emptyset, \emptyset, \texttt{false}) & \text{otherwise} \end{cases}$
        if $a \in \mathscr{A}_{basic}$ then
            HANDLEBASICACTIVITY($a$,$v_e$)
        else if $a \in \mathscr{A}_{flow}$ then
            HANDLEFLOW($a$,$v_e$)
        else if $a = \mathsf{process}$ then
            // Directly handle the contained activity
            HANDLEACTIVITY($a'$,$v_e$), $(a,a') \in \mathsf{HR}$
            $\mathsf{writes}_\bullet(a) \leftarrow \mathsf{writes}_\bullet(a')$
        end if
        for all $l \in \mathscr{L}_{out}(a)$ do
            HANDLELINK($l$,$v_e$)
        end for
    end if
end procedure
```

gorithm 2. It checks whether all incoming links of the currently visited activity were handled, and whether the parent activity was visited. If not, it returns, since the activity will be reached via all non-visited links or via the parent activity again. Note that BPEL does not allow control links to form a cycle. Thus, all incoming links of an activity can always be visited before the activity itself. If all incoming links (in case there are any) and the parent activity (in case there is any) were handled, the information of the predecessors of the activity has to be put into $\mathsf{writes}_\circ$. If the activity has incoming links, their information has to be joined, since a real process execution reaches the activity by these links. The joining is done by the function joinLinks, which is explained in the next section 3.4. If the activity has no incoming links, it may have a parent in the hierarchy tree. If that is the case, the information of $\mathsf{writes}_\circ$ of the parent has to be copied, since the current activity will be started right after the parent activity and thus has the data of the parent activity available. If the current activity has no incoming links and no parent activity (i.e., it is the `process` element itself), $\mathsf{writes}_\circ$ is initialized to contain no writers. After $\mathsf{writes}_\circ$ has been determined, the activity itself is handled by the algorithms for handling the respective types of activities: HANDLEBASICACTIVITY (section 3.5) to handle basic activities and HANDLEFLOW (section 3.6) to handle the flow activity. If other activity types have to be handled, the respective function has to be called here. After handling the activity itself, its outgoing links are traversed by HANDLELINK which in turn calls HANDLEACTIVITY for the target of each link. It is important to note that this is the same order as a BPEL engine executes the activities in a BPEL process.

3.4. Handling Incoming Links

The function $\mathsf{joinLinks} : (a, v_e) \mapsto (P, D, d)$ is used to join the information on the incoming links in Algorithm 2. P, D, and d are defined as follows:

The Set P of Possible Writers Case (1) in Algorithm 3 handles the situation in which the join condition cannot re-enable any writes. It contains two subcases: (i) The join condition is a logical AND over all incoming links, (ii) there is only one incoming link and the join condition does not negate the status of the incoming link. (i) If the join condition is a logical AND over all incoming links, and during process execution the status of at least one incoming link is set to `false`, the activity itself is not executed. Thus, any disabled writers cannot be re-enabled. A re-enablement may happen later as illustrated on activity $\mathsf{r}_2^{\mathsf{x}}$ in figure 1. (ii) If there is only one incoming link, the current activity can only be reached over that path. If the join condition does not negate the status of the incoming link, the target activity is executed iff the status of the incoming link is `true`. In other words, if the status of the incoming link is `false`, the activity itself is not executed and thus any disabled writers cannot be re-enabled at this activity. If the status of the incoming link is `true`, the disabled writers cannot be re-enabled either, since there is no alternative execution path reaching the activity. If the join condition negates the status of the incoming link, the

Algorithm 3 Determining the set P of possible writers

```
if $(|\mathscr{L}_{in}(a)| = 1 \wedge \neg\mathsf{negateslinkstatus}(\mathsf{jc}(a))) \vee \mathsf{jc}(a)$
is a logical AND over all incoming links then
    $P \leftarrow \bigcup_{l \in \mathscr{L}_{in}(a)} \mathsf{poss}_\bullet(l, v_e)$        (1)
else
    $P \leftarrow \bigcup_{l \in \mathscr{L}_{in}(a)} \mathsf{poss}_\bullet(l, v_e) \cup \mathsf{dis}_\bullet(l, v_e) \setminus \bigcap_{l \in \mathscr{L}_{in}(a)} \mathsf{dis}_\bullet(l, v_e)$        (2)
end if
```

target activity is executed iff the status of the incoming link is `false`. As a consequence, disabled writers may be re-enabled at the target activity. We use the function $\mathsf{negateslinkstatus} : \mathscr{C} \to \mathbb{B}$ to return `true` iff the given join condition negates the status of its incoming link.

Case (2) handles the case where the join can enable disabled writes. An active incoming link not contained in the possible dead path from a last possible writer can set the state of the current activity to active. Therefore, the data is not taken from the dead writer, but from a preceding write disabled by the dead writer. For example, this is the case at $\mathsf{r}_2^{\mathsf{x}}$ in figure 1, where $\mathsf{w}_1^{\mathsf{x}}$ has to be enabled again.

All in all, disabled writes get enabled again at a join activity that does not have an AND join condition. The only exception are writes that are dead during the execution on all paths reaching the current activity. These writes cannot be enabled again by an incoming path and thus are not put into P.

The Set D of Disabled Writers The Algorithm 4 for determining D is similar to the algorithm determining P, but handles disabled writers instead of enabled ones. If the join cannot re-enable any writers (case (1)), the set of disabled writers remains the same. If the join enables disabled writers (case (2)), D is the set of writers that are not re-enabled.

Algorithm 4 Determining the set D of disabled writers

```
if $(|\mathscr{L}_{in}(a)| = 1 \wedge \neg\mathsf{negateslinkstatus}(\mathsf{jc}(a))) \vee \mathsf{jc}(a)$ is a logical AND over all incoming links then
    $D \leftarrow \bigcup_{l \in \mathscr{L}_{in}(a)} \mathsf{dis}_\bullet(l, v_e)$    (1)
else
    $D \leftarrow \bigcap_{l \in \mathscr{L}_{in}(a)} \mathsf{dis}_\bullet(l, v_e)$    (2)
end if
```

State "Dead" of an Activity (d) An activity a may be dead on a path from any directly preceding writer (or from the root node if there is no preceding writer) if the join condition of a evaluates to `false`.

The decision logic presented in section 3.5 uses the value of $\mathsf{mbd}_\circ(a, v_e)$ to decide whether possible writers get disabled or invalid: If $\mathsf{mbd}_\circ(a, v_e)$ is `true`, then the possible writers get disabled. Otherwise, they get invalid. An invalid writer is never revived due to dead path elimination. Therefore, the set of possible writers decreases if more writers get invalid. On the other hand, a writer may never be treated as "invalid" if it can be a possible writer. Therefore, a one-sided error is acceptable: $\mathsf{mbd}_\circ(a, v_e)$ may return `true`, even if it should be `false`. But $\mathsf{mbd}_\circ(a, v_e)$ may never return `false`, if it should return `true`.

We use following approximation to determine $\mathsf{mbd}_\circ(a, v_e)$:

i) If the join condition always evaluates to `true`, $\mathsf{mbd}_\circ$ is set to `false`.
ii) If the join condition contains negations, $\mathsf{mbd}_\circ$ is set to `true`.
iii) Otherwise, the join condition is evaluated with the negated values of $\mathsf{mbd}_\bullet$ of each incoming link.

After presenting details of each case, we will show an example illustrating the one-sided error of this approximation of $\mathsf{mbd}_\circ(a, v_e)$ at the end of this section.

For case i we define the function $\mathsf{alwaystrue} : \mathscr{C} \to \mathbb{B}$ to return `true` iff the given condition always evaluates to `true`. The function may have a one-sided error: If the join condition always evaluates to `true`, $\mathsf{alwaystrue}$ may return `false`, but not the other way round. One implementation of $\mathsf{alwaystrue}$ is to check whether the given condition equals the string `true()`. This implementation runs in linear time in the length of the condition. In the general case, the implementation is in NP: Join conditions are Boolean formulas over the status of the incoming links. Thus, checking for always evaluating to `true` is equal to check for satisfiability of the negation of the formula. Since the satisfiability problem is in NP [3], $\mathsf{alwaystrue}$ is also in NP.

For case ii, we define the function $\mathsf{negations} : \mathscr{C} \to \mathbb{B}$ to return `true`, iff the given join condition contains negations.

For case iii, we define the semantics operation $[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$. It takes the negated value of $\mathsf{mbd}_\bullet(l, v_e)$ as the current status of each link l in the join condition of the given activity a and evaluates the join condition. This ensures proper handling of the activity with respect to the directly preceding writers: The join condition of a does not include any negations. Due to the definition of $\mathsf{mbd}_\bullet$ on links, $\mathsf{mbd}_\bullet(l, v_e)$ is `true` if l may be dead. If $\mathsf{mbd}_\bullet(l, v_e)$ is `false`, the link is surely not dead from any path from all directly preceding writers (or the root node if there is no directly preceding writer). By using the negated value of $\mathsf{mbd}_\bullet(l, v_e)$, the status of the link is reflected: If a link is dead, the status of the link is `false` during process execution. If the link is not dead, the status of the link is `true`. Note that by using the negated value of $\mathsf{mbd}_\bullet(l, v_e)$ we evaluate the join condition in the case in which most links are dead in one process execution.

We illustrate the use of $[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$ by the process presented in figure 3, which is a modified version of the process presented in figure 1. $\mathsf{w}_1^{\mathsf{x}}$ and $\mathsf{w}_2^{\mathsf{y}}$ write to different variables x and y. Furthermore $\mathsf{w}_3^{\mathsf{x,y}}$ writes to both x and y. $\mathsf{r}_1^{\mathsf{x,y}}$ is reading from both x and y. l_i are

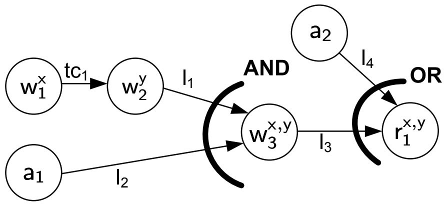


Figure 3. Process illustrating $[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$

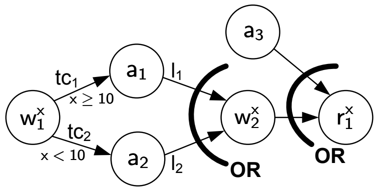


Figure 4. Illustration of the one-sided error

links with the default transition condition `true`. $\mathsf{w}_1^{\mathsf{x}}$ and $\mathsf{w}_3^{\mathsf{x,y}}$ both write to x. Because of the transition condition tc_1 and the AND join on $\mathsf{w}_3^{\mathsf{x,y}}$, $\mathsf{w}_3^{\mathsf{x,y}}$ can be dead and the value of $\mathsf{w}_1^{\mathsf{x}}$ reaches $\mathsf{r}_1^{\mathsf{x,y}}$ in that case. On the other hand, the value written by $\mathsf{w}_2^{\mathsf{y}}$ never reaches $\mathsf{r}_1^{\mathsf{x,y}}$. If $\mathsf{w}_2^{\mathsf{y}}$ is dead, $\mathsf{w}_3^{\mathsf{x,y}}$ is dead, too. If $\mathsf{w}_2^{\mathsf{y}}$ is not dead, $\mathsf{w}_3^{\mathsf{x,y}}$ is not dead either.

Note that we cannot use $[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$ to handle case ii, where the join condition contains negations. Assume that the negation in a join condition of an activity b is $\neg l$. Thus, the join condition negates l. Furthermore, assume that $\mathsf{mbd}_\bullet(l, v_e) = \mathtt{true}$. Recall that $\mathsf{mbd}_\bullet(l, v_e) = \mathtt{true}$ denotes that there exists a path from the root node to the current node, which sets the status of the link l to `false`, because of DPE. $\mathsf{mbd}_\bullet(l, v_e) = \mathtt{true}$ does not state that l is dead in all cases: It is possible, that there are paths, where the status of the link is `true`. In a process execution, the activity b is executed if and only if the status of the incoming link l evaluates to `false`. Otherwise, the activity b is dead.

$$[\![b, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet} = \underbrace{\neg(\underbrace{\neg\mathsf{mbd}_\bullet(l, v_e)}_{l})}_{\text{join condition}} = \neg(\neg(\mathtt{true})) = \mathtt{true}$$

The negated value of $[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$ is used in Algorithm 5 to determine d. In our case, $\neg[\![b, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet} = \neg\mathtt{true} = \mathtt{false}$ and thus d would be set to `false`, which means that activity b is never dead. This contradicts the fact that the activity b is executed iff the status of the incoming link l evaluates to `false`. Thus, $[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$ cannot be used if the join condition of an activity contains negations.

Algorithm 5 presents the algorithmic summary of the determination of d. The used approximation only considers the join condition of a and $\mathsf{mbd}_\bullet(l, v_e)$ of each incoming link of a. It does not take other information in account, which may lead to a wrong value of $\mathsf{mbd}_\circ$: Consider the process illustrated in figure 4. Since there is a transition condition on tc_1, $\mathsf{mbd}_\bullet(\mathsf{tc}_1, v_e) = \mathtt{true}$ as illustrated in section 3.7. This propagates to l_1, where $\mathsf{mbd}_\bullet(\mathsf{l}_1, v_e) = \mathtt{true}$. Similar for tc_1 and l_2. Since one of the incoming links at $\mathsf{w}_2^{\mathsf{x}}$ may be dead, $\mathsf{w}_2^{\mathsf{x}}$ itself is considered as possibly dead ($\mathsf{mbd}_\circ(\mathsf{w}_2^{\mathsf{x}}, v_e) = \mathtt{true}$). Thus, $\mathsf{w}_2^{\mathsf{x}}$ is considered to disable $\mathsf{w}_1^{\mathsf{x}}$. However, $\mathsf{w}_1^{\mathsf{x}}$ revives at $\mathsf{r}_1^{\mathsf{x}}$, because of the OR join. If the whole processing history is taken into consideration, $\mathsf{mbd}_\circ(\mathsf{w}_2^{\mathsf{x}}, v_e)$ has to be set to `false`: The control flow always reaches $\mathsf{w}_2^{\mathsf{x}}$ if $\mathsf{w}_1^{\mathsf{x}}$ is executed, since the transition conditions tc_1 and tc_2 are mutual exclusive.

Algorithm 5 Determining whether an activity may be dead

$$d \leftarrow \begin{cases} \mathtt{false} & \mathsf{alwaystrue}(\mathsf{jc}(a)) \\ \mathtt{true} & \mathsf{negations}(\mathsf{jc}(a)) \\ \neg[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet} & \text{otherwise} \end{cases}$$

3.5. Handling Basic Activities

In this section, we describe the handling for basic activities. The term 'writing activity' is used to refer to activities that can write data (`receive`, `assign`, ...). To define $\mathsf{writes}_\bullet(a, v_e)$, the activity a is checked whether it writes to the given variable element. We define the function $w : \mathscr{A} \times E^{Q_\mathscr{V}} \to \mathbb{B}$ to state whether the given activity completely writes to the given variable element. If an activity a changes a part of a variable element v_e and not the whole variable $w(a)$ returns `false`. If an activity a writes to a parent of v_e, $w(a)$ returns `true`, since the given query completely changes v_e.

If a is not a writing activity (`empty`, `wait`, ...), $\mathsf{writes}_\bullet(a, v_e)$ is the identity of $\mathsf{writes}_\circ(a, v_e)$ (Case (1) in Algorithm 6).

If a is a writing activity ($w(a, v_e) = \mathtt{true}$) the result depends on the value of $\mathsf{mbd}_\circ(a, v_e)$. If $\mathsf{mbd}_\circ(a, v_e) = \mathtt{true}$, the possible writes are added to the disabled writes and a is put as the only writer (Case (2) in Algorithm 6). Recall that $\mathsf{mbd}_\circ(a, v_e) = \mathtt{true}$ denotes that a may not be executed due to DPE on a path from any directly preceding writer to a (or from the root node if there is no directly preceding writer). Assume w to be a valid preceding writer for an activity a. At the activity a itself, the write of a is the only valid write, since a definitely overwrites the write of w if a was exe-

Algorithm 6 Handling a basic activity

```
procedure HandleBasicActivity($a$,$v_e$)
    $\mathsf{writes}_\bullet(a,v_e) \leftarrow$
    $\begin{cases} \mathsf{writes}_\circ(a,v_e) & \text{if } \neg w(a) \quad (1) \\ (\{a\}, \mathsf{dis}_\circ(a,v_e) \cup \mathsf{poss}_\circ(a,v_e), \texttt{false}) & \text{if } w(a,v_e) \wedge \mathsf{mbd}_\circ(a,v_e) \quad (2) \\ (\{a\}, \mathsf{dis}_\circ(a,v_e), \texttt{false}) & \text{if } w(a,v_e) \wedge \neg\mathsf{mbd}_\circ(a,v_e) \quad (3) \end{cases}$
end procedure
```

cuted. If w was executed, a is not necessarily executed ($\mathsf{mbd}_\circ(a,v_e) = \texttt{true}$) and thus a may not have overwritten the value written by w. Since w can become valid again at a successor of a, w has to be stored.

In the other case, where $\mathsf{mbd}_\circ(a,v_e) = \texttt{false}$, the possible writes can never be active again and are removed from the set of possible writers and not added to the set of disabled writers (Case (3) in Algorithm 6). Assume w being a valid preceding writer for an activity a. If w was executed, a will also be executed ($\mathsf{mbd}_\circ(a,v_e) = \texttt{false}$). If w was not executed, a will not be executed either. Thus, the write of w is always overwritten by the write of a and there is no need to store w.

In both cases (2) and (3), a is a writer to v_e. In this case, a is the starting point of all paths from a to a subsequent writer. Since a is not dead at the beginning of all paths starting at a, $\mathsf{mbd}_\bullet = \pi_3(\mathsf{writes}_\bullet)$ is set `false` in cases (2) and (3) in Algorithm 6. It is important to note that Algorithm 5 and Algorithm 8 ensure that $\mathsf{mbd}_\circ$ and $\mathsf{mbd}_\bullet$ are set to `true` if the activity or the link may be dead and thus the write of a can survive.

3.6. Handling a Flow Activity

The traversal of the activities nested in a flow activity starts from the roots of the flow activity. It is important to note that links may cross the boundary of a flow activity. Therefore, a root of a flow activity is an activity in the flow with no incoming links from any activities inside the flow. The outgoing links of the roots are traversed in HandleActivity and thus all activities in the flow get visited. As soon as HandleActivity returns, $\mathsf{writes}_\bullet$ is defined for all activities contained in the flow. The information of the flow's leaf activities has to be joined into the flow's $\mathsf{writes}_\bullet$. A leaf of a flow is an activity with no outgoing links to any other activity inside the flow. It is important to note that $\mathsf{mbd}_\bullet$ is not constructed by using $\mathsf{mbd}_\bullet$ of the leaves, but by using the value of $\mathsf{mbd}_\circ$ of the flow: If a flow activity itself is not dead

Algorithm 7 Handling a flow activity

```
procedure HandleFlow($f$,$v_e$)
    $roots \leftarrow \{a \mid a \in \mathsf{children}(f) \wedge$
            $\neg\exists l : (a', l, a) \in \mathsf{LR}, a' \in \mathsf{descendants}(f)\}$
    for all $a_r \in roots$ do
        HandleActivity($a_r$,$v_e$)
    end for
    $leaves \leftarrow \{a \mid a \in \mathsf{children}(f) \wedge$
            $\neg\exists l : (a, l, a') \in \mathsf{LR}, a' \in \mathsf{descendants}(f)\}$
    $\mathsf{writes}_\bullet(f,v_e) \leftarrow \big(\bigcup_{a_l \in leaves} \mathsf{poss}_\bullet(a_l,v_e),$
            $\bigcup_{a_l \in leaves} \mathsf{dis}_\bullet(a_l,v_e), \mathsf{mbd}_\circ(f)\big)$
end procedure
```

at the beginning of its execution, it remains not dead at the end of the execution, even if all leaves of a flow activity are dead. The complete handling is presented in Algorithm 7.

3.7. Handling Links

A control link has exactly one source activity and one target activity. Therefore, the analysis result of the source activity can be directly taken as $\mathsf{writes}_\circ$. Furthermore, a link l cannot perform a write to a variable but is associated with a transition condition, which may evaluate to `false`. The subsequent activity may be dead, if the source of the link may be dead or if the transition condition on link l may evaluate to `false`.

A special case is when a transition condition always evaluates to `true`. In this case, the value of $\mathsf{mbd}_\circ$ is copied to $\mathsf{mbd}_\bullet$, since the transition condition has no influence on whether the subsequent activity may be dead. In general, it is not possible to check whether a transition condition always evaluates to `true`, since the satisfiability of XPath expressions is undecidable [2]. Nevertheless, we reuse the function alwaystrue presented in section 3.4, but we demand that it returns `false` if it cannot determine whether the given XPath expression is satisfiable. One implementation for alwaystrue is presented in section 3.4: Check whether the given string equals `true()`. Using this implementation, the algorithm produces an over-approximation, as illustrated in section 3.4.

After $\mathsf{writes}_\bullet(l,v_e)$ is set, the link is marked as visited and the target activity is visited. The complete handling is presented in Algorithm 8.

3.8. Explicit Data Links

After AnalyzeProcessModel completed, $\mathsf{writes}_\circ$ and $\mathsf{writes}_\bullet$ are defined for all variable elements. $\mathsf{poss}_\circ(a,v_e) = \pi_1(\mathsf{writes}_\circ(a,v_e)$ returns the set of

Algorithm 8 Handling a link

```
procedure HandleLink($l$,$v_e$)
    $a, a' : (a, l, a') \in \mathsf{LR}$
    $\mathsf{writes}_\circ(l, v_e) \leftarrow \mathsf{writes}_\bullet(a, v_e)$
    $\mathsf{writes}_\bullet(l, v_e) \leftarrow$
        $\begin{pmatrix} \mathsf{poss}_\circ(l, v_e), \\ \mathsf{dis}_\circ(l, v_e), \\ \mathsf{mbd}_\circ(l, v_e) \vee \neg \mathsf{alwaystrue}(\mathsf{tc}(l)) \end{pmatrix}$
    $\mathsf{visited}(l) \leftarrow \mathtt{true}$
    HandleActivity($a'$,$v_e$)
end procedure
```

possible writers for each given activity and variable element. As shown in section 3.1, a read of an activity a on a variable element v_e ($r(a, v_e) = \mathtt{true}$) also has to read all data written to children of v_e. A data link is a tuple (w, z), where w is an activity writing data and z an activity or a link possibly reading the data written by w. Thus, the set of all data links in the given BPEL process $\mathsf{DL} \subset \wp(\mathscr{A} \times \mathscr{A} \cup \mathscr{L})$ is defined as follows:

$$\mathsf{DL} := \{(w, z) \mid z \in \mathscr{A} \cup \mathscr{L}, w \in \mathsf{poss}_\circ(z, v'_e), \\ v'_e \sqsubseteq v_e, r(z, v_e) = \mathtt{true}\}$$

This paper provides data links for analysis purposes. Using them to replace BPEL's shared variable behavior at runtime requires further specification of runtime semantics.

3.9. Algorithm Applied

Table 2 presents the result of the data-flow algorithm applied to the example process of figure 1 having jc_1 set to AND. Note that the process of figure 1 contains a single variable x of a simple type. $\mathsf{poss}_\circ(\mathsf{r}_1^\mathsf{x}) = \{\mathsf{w}_2^\mathsf{x}\}$ denotes that w_2^x is the only possible writer for r_1^x.

The complexity of the algorithm depends on the implementation of alwaystrue and the checking of "jc is a logical AND over all incoming links". In the general case, alwaystrue cannot be implemented. If the requirements on the implementation are lowered, both alwaystrue and the checking for an AND join run in linear time in the number of links. Then, the limiting factor of the algorithm is the set of possible and the set of disabled writers. In the worst case, the flow is a graph where the activities are ordered sequentially and each link takes a transition condition. Then, $\mathsf{poss}_\bullet(l, v_e)$ contains all predecessors and has to be copied to $\mathsf{poss}_\circ(a, v_e)$ at each activity. For a graph with n activities, this happens $0 + 1 + \cdots + n - 2 + n - 1 = \frac{1}{2}n(n-1) = \mathscr{O}(n^2)$ times. Since the number of the activities is less than the number of links, the complexity of the analysis the process for one v_e is $\mathscr{O}(n^2)$, where n is the number of links in the flow activity. Since the depth-first search is started for each $v_e \in E^{Q_\mathscr{V}}$, the overall complexity is $\mathscr{O}(|E^{Q_\mathscr{V}}| \cdot n^2)$.

Table 2. Analysis result for the example process of figure 1 with jc_1 set to AND

Activity / Link	Possible Writers ($\mathsf{poss}_\circ$)	Disabled Writers ($\mathsf{dis}_\circ$)	May Be Dead ($\mathsf{mbd}_\circ$)
w_1^x	$\emptyset$	$\emptyset$	`false`
link with tc_1	$\{\mathsf{w}_1^\mathsf{x}\}$	$\emptyset$	`false`
w_2^x	$\{\mathsf{w}_1^\mathsf{x}\}$	$\emptyset$	`true`
l_1	$\{\mathsf{w}_2^\mathsf{x}\}$	$\{\mathsf{w}_1^\mathsf{x}\}$	`false`
a_1	$\emptyset$	$\emptyset$	`false`
l_2	$\emptyset$	$\emptyset$	`false`
r_1^x	$\{\mathsf{w}_2^\mathsf{x}\}$	$\{\mathsf{w}_1^\mathsf{x}\}$	`false`
l_3	$\{\mathsf{w}_2^\mathsf{x}\}$	$\{\mathsf{w}_1^\mathsf{x}\}$	`false`
a_2	$\emptyset$	$\emptyset$	`false`
l_4	$\emptyset$	$\emptyset$	`false`
r_2^x	$\{\mathsf{w}_1^\mathsf{x}, \mathsf{w}_2^\mathsf{x}\}$	$\emptyset$	`false`

4. Related Work

Current data-flow analysis algorithms do not explicitly deal with "graph-based programs", but deal with traditional "structured programs". Thus, current approaches that take BPEL as input treat control links as a special case: The work of [12] transforms BPEL into a Concurrent Static Single Assignment Form (CSSA, [10]) representation. The complexity of this transformation is $\mathscr{O}(|V| \cdot n^2)$, where V denotes the set variables and n denotes the number of nodes. In the case of sequential execution of activities in a flow activity, their algorithm returns too many possible writers. For example, if their algorithm is applied to the example presented in figure 1 with jc_1 being an AND, the algorithm returns $\{\mathsf{w}_1, \mathsf{w}_2\}$ as the set of possible writers for r_1^x. Our algorithm is more precise and returns w_2^x as the only possible writer for r_1^x. Their algorithm treats each activity with incoming links as an `if` statement with the join condition as branching condition. The activity is executed if the branching condition evaluates to `true`. An artificial joining node (called "Phi-node") after the `if` statements joins the information from both paths. As a result, each activity in a flow can be skipped in the abstract interpretation, regardless of the structure of the graph. This does not reflect the idea of dead path elimination, where the dead status propagates through the graph. The work of [5] is based on the work of [14] and provides data-flow equations for BPEL activities. However, it does not consider transition conditions, join conditions and dead path elimination. Both [5] and [12] do not deal with complex types and thus offer room for improvement besides dead path elimination.

Full technical details of the algorithm are described in our technical report [9]. In addition, the technical report describes the transformation of $\mathsf{poss}_\circ$ to the required input of the splitting algorithm described in [7].

It is shown in [2] that it is undecidable whether a given XPath expression is satisfiable. Since standard BPEL allows arbitrary XPath expressions to be used as transition conditions, it follows immediately that an exact determination of data links in BPEL processes is undecidable. Nevertheless, the result of our algorithm is an improvement in comparison to other approaches, since the algorithm strictly follows the semantics of DPE and can handle complex types.

5. Conclusion and Future Work

In this paper, we motivated that algorithms for data-flow analysis on BPEL processes should be aware of dead path elimination. Our presented algorithm allows for determining data links in BPEL processes, where dead path elimination is activated. We showed how join conditions can be used to reduce the number of possible writers for an activity.

HANDLEACTIVITY allows for adding handling of arbitrary structured activities besides the described `flow` activity. Adding handling of other structured activities is part of our ongoing work. Of particular interest are fault handlers designed for catching a `joinFailure` which is thrown if dead path elimination is not active. Thus, future work will provide a complete algorithm to determine data links in arbitrary BPEL processes.

Acknowledgments. We thank Daniel Martin, Ralph Mietzner, Simon Moser, and Dirk Nowotka for their valuable input. We also would like to thank our colleagues in the IAAS and ISTE institutes at the University of Stuttgart for reviewing earlier drafts. Oliver Kopp is funded by the German Federal Ministry of Education and Research (project Tools4BPEL, project number 01ISE08).

References

[1] A. V. Aho et al. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[2] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems – PODS '05*, pages 25–36. ACM, 2005. doi:10.1145/1065167.1065172.

[3] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing (STOC '71)*, pages 151–158. ACM, 1971. doi:10.1145/800157.805047.

[4] F. Curbera et al. Exception Handling in the BPEL4WS Language. In *Business Process Management – International Conference (BPM 2003)*, volume 2678 of *LNCS*, pages 276–290. Springer, 2003. doi:10.1007/3-540-44895-0_19.

[5] T. Heidinger. Statische Analyse von BPEL4WS-Prozemodellen, 2003. Studienarbeit, Humboldt-Universität zu Berlin.

[6] R. Khalaf. *Supporting Business Process Fragmentation While Maintaining Operational Semantics – A BPEL Perspective*. Doctoral Thesis, Universität Stuttgart, 2008.

[7] R. Khalaf, O. Kopp, and F. Leymann. Maintaining Data Dependencies Across BPEL Process Fragments. In *Service-Oriented Computing – ICSOC 2007*, volume 4749 of *LNCS*, pages 207–219. Springer, 2007. doi:10.1007/978-3-540-74974-5_17.

[8] R. Khalaf and F. Leymann. Role-based Decomposition of Business Processes using BPEL. In *ICWS 2006*, pages 770–780. IEEE Computer Society, 2006. doi:10.1109/ICWS.2006.56.

[9] O. Kopp, R. Khalaf, and F. Leymann. Reaching Definitions Analysis Respecting Dead Path Elimination Semantics in BPEL Processes. Technical Report 2007/04, Universität Stuttgart, IAAS, 2007.

[10] J. Lee, S. P. Midkiff, and D. A. Padua. Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In *Proceedings of The 10th International Workshop on Languages and Compilers for Parallel Computing (LCPC '97)*, volume 1366 of *LNCS*, pages 114–130. Springer, 1998. doi:10.1007/BFb0032687.

[11] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 33(2):326–348, 1994.

[12] S. Moser et al. Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis. In *Proceedings of IEEE International Conference on Services Computing (SCC 2007)*, pages 98–105, July 2007. doi:10.1109/SCC.2007.22.

[13] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2004.

[15] OASIS. *Web Services Business Process Execution Language Version 2.0 – OASIS Standard*, 2007.

[16] C. Ouyang et al. Formal Semantics and Analysis of Control Flow in WS-BPEL (Revised Version). BPM Center Report BPM-05-15, BPMcenter.org, 2005.

[17] W3C. XML Path Language (XPath) Version 1.0. W3C Recommendation, 1999.