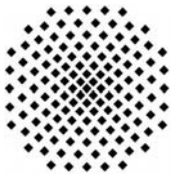

Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

Reaching Definitions Analysis Respecting Dead Path Elimination Semantics in BPEL Processes

Oliver Kopp, Rania Khalaf,
Frank Leymann

Report 2007/04
November 2007

Institut für Architektur von Anwendungssystemen

Universitätsstraße 38
70569 Stuttgart
Germany

CR: F.3.2, H.4.1

1 Introduction

The Business Process Execution Language for Web Services (BPEL) is a workflow language geared towards Service Oriented Computing. BPEL provides basic workflow capabilities, such as the ability to impose control on a set of activities via explicit links, as well as advanced features such as recovery and event handling. The area of workflow often requires understanding the data dependencies as well as the control dependencies between activities. This aids in business process design as well as in analysis and rengineering. BPEL, however, uses shared variables to model data. Activities read and write data to these variables, i.e. there is no explicit data link construct. In order to draw out the flow of data between the activities of a BPEL process, one must therefore perform data-flow analysis on that process.

To address this problem, we present an algorithm that statically determines such data dependencies. "Statically" means, that we do not determine the dependencies per process instance at runtime, but on the process model without creating any instances. Mainstream data-flow analysis techniques are presented in [ALSU06, Muc97, NNH04]. However, these techniques cannot be directly applied to a BPEL process, since BPEL supports both parallelism and dead path elimination (DPE, [CDG+03, OAS07, CKLW03, BK05, LR00]). DPE is a technique used in BPEL to propagate activity disablement down paths that can no longer be executed. The algorithm in this paper consists of reaching definitions analysis dealing with DPE, enabling it to reduce the number of data dependencies when compared to approaches not dealing with DPE.

While there may be many varied and additional uses of the presented data-flow analysis, our main motivation for creating a data-flow analysis algorithm for BPEL was a need to provide results that enable splitting BPEL processes based on business need: 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 unsplit process are maintained. The splitting of BPEL processes was introduced in [KL06]. In that work, data dependencies were dealt with by using an extension of BPEL, known as BPEL-D, which enables a user to define explicit data links instead of BPEL's shared variables. Data links are based on def-use edges (also known as writer-reader-relations), but also have runtime semantics. However, when splitting a standard BPEL process, instead of a BPEL-D process, the results of a BPEL-based data-flow analysis algorithm are needed.

We extended the work of [KL06] in [KKL07] to split standard BPEL processes. One of the inputs to the splitting algorithm was a generic result that could be achieved from an arbitrary data-flow analysis algorithm. Data-flow analysis algorithms are selected based on the problem space one is trying to address. In our case, the focus is on using the results in order to exchange the data between fragments of the process after it has been partitioned by a process designer. A data dependency between two activities in different fragments will be roughly translated into a message sent from one fragment to another. Therefore, it is important to return as few data dependencies as possible in

order to minimize the communication overhead between the partners. Our work here relates to [KKL07] in that it provides one specific data-flow analysis algorithm that reduces the number of data dependencies due to its handling of DPE.

The splitting algorithms, defined across [KL06], [KKL07] [KL07], place a set of restrictions on BPEL. These restrictions are therefore also used in this work. The restrictions are:

- Flow, scope, and while activities are the only structured activities.
- Dead path elimination is always activated.
- Writers may not write in parallel to a reader of a variable. This is ensured, if the Bernstein Criterion holds for the BPEL process [LA94].
- Event handlers are disallowed.
- Termination handlers are disallowed.
- Fault handlers get their data from the coordinator (cf. [KL07]).
- Compensation handlers have BPEL 1.1 [CDG+03] semantics: The compensation handler takes a snapshot of the scope as soon as the compensation handler gets installed, which is when the scope completes.

In section 2 we present the challenges of DPE for data-flow analysis and in section 3, the state of the art in the field of data-flow analysis on BPEL processes. Afterwards, we present a reaching definitions analysis algorithm for BPEL 1.1 [CDG+03] and BPEL 2.0 [OAS07] processes following the restrictions listed above and returning less data dependencies than existing algorithms (Section 4). Appendix A presents an example process and shows the results of the reaching definitions analysis. Section 5 shows the algorithms needed to construct the input format for the algorithm presented in [KKL07]. Finally, section 6 provides a summary of the key contributions of the presented algorithm and gives an outlook on future work.

2 The Challenges of DPE for Data-flow Analysis

In order to address data-flow analysis in BPEL, we present a short summary of the behavior of its 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, with each link associated with a 'transitionCondition'. 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 disabled (not executed) and the status of its outgoing links is set to `false`. The default value for a transition condition is `true`, and for the join condition it is the disjunction of the status of all incoming links. Regardless of whether the activity is executed or not, the outgoing links are fired and the target activities visited.

It should be noted that the existence of multiple incoming links associated with an activity always represents a synchronizing join in BPEL. Dead path elimination (DPE) is the technique of propagating the disablement of an activity so that activities downstream do not wait forever for its completion. The propagation is needed, because each activity carries a join condition which is evaluated on the status of the incoming link. In this section, we will show aspects of DPE making data-flow analysis non-trivial. For a detailed explanation of DPE see [CKLW03].

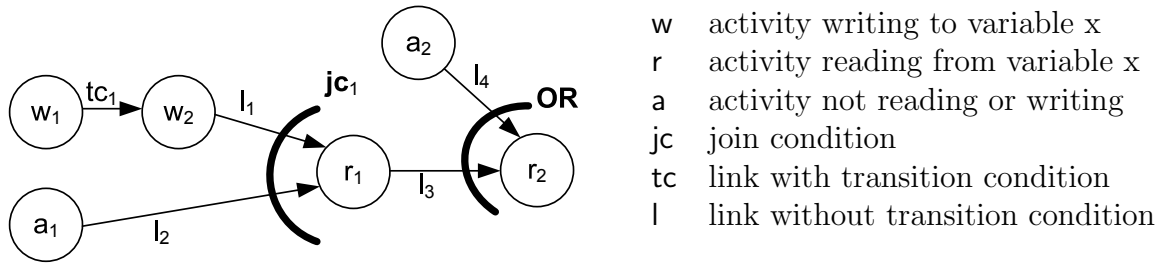

Figure 1: A process illustrating the challenges of DPE

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

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

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

3 Related Work

Current data-flow analysis algorithms do not treat graph-based programs as a new concept, but as a special case of structured programming languages. Thus, the current approaches take BPEL as input and treat control links as a special case. The work of [MMG+07] transforms BPEL into a Concurrent Static Single Assignment Form (CSSA, [LMP97]) representation. 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. Our algorithm is more precise and returns w_2 as the only possible writer for r_1. Their algorithm treats each activity with incoming links as an if statement with the join condition as condition. The activity is executed if the condition evaluates to `true`. An artificial joining node (called "Phi-node") after the if statements joins the information from both paths. Thus, every activity in a flow can be skipped. This does not reflect the idea of dead path elimination, where the dead status propagates through the graph. The work of [Hei03] is based on the work of [CC76] and provides data-flow equations for BPEL activities. However, it does not consider transition conditions, join conditions and dead path elimination. Both [MMG+07] and [Hei03] do not deal with complex types and thus offer room for improvement besides dead path elimination.

[BFG05] showed 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 dependencies in BPEL processes is undecidable. The result of our algorithm is an improvement in comparison to other approaches, since the algorithm follows the semantics of DPE.

4 The Algorithm

We presented the challenges of dead-path elimination for data-flow analysis in section 2. In this section, we present an algorithm which realizes a reaching definition analysis on BPEL processes and respects dead-path elimination behavior.

To stay close to existing syntax formalizations of BPEL, our notation is based on the one presented in [OVA+05], where a detailed explanation of the notation can be found. The used notation is presented in tables 1, 2 and 3. The idea of the formalization is to connect nested elements with the HR relation. The child a of a while loop w is connected to w with a condition c: $(w, c, a) \in \mathsf{HR}$. The children of a flow activity f are connected with $\perp$ to f, since the children of a flow activity are nested without any condition. The links in the flow activity are explicitly modeled by the set $\mathcal{L}$. A compensation handler is connected with a compensation event to a scope. Finally, the fault handler is connected with a fault event to the scope.

Notation	Meaning
$\mathcal{A}_{basic}$	The set of all basic activities
$\mathsf{process}$	The process element
$\mathcal{A}_{flow}$	The set of all flow activities
$\mathcal{A}_{scope}$	The set of all scope activities. $\mathsf{process} \in \mathcal{A}_{scope}$
$\mathcal{A}_{while}$	The set of all while activities
$\mathcal{A} = \mathcal{A}_{basic} \cup \mathcal{A}_{flow} \cup \mathcal{A}_{scope} \cup \mathcal{A}_{while}$	The set of all activities
$\mathcal{C}$	Set of all Boolean conditions
$\pi_i(t)$	Returns the projection to the i^{th} component of a tuple t
$\mathsf{HR} \subseteq \mathcal{A} \times \mathcal{B} \times \mathcal{A}$	The hierarchy relation denoting the nesting of activities. Let $h = (a, c, a') \in \mathsf{HR}$. Then a is a parent of a'. The execution order of activities nested in a flow activity is specified by the (control) links $\mathcal{L}$.
$\mathcal{L}$	All (control) links (i.e., edges in flow activities)
$\mathsf{LR} = \mathcal{A} \times \mathcal{L} \times \mathcal{A}$	The control link relation
$\wp(S)$	Denotes the power set of a set S.
$\mathcal{L}_{in} : \mathcal{A} \to \wp(\mathcal{L})$	Returns the set of all incoming links of an activity.
$\mathcal{L}_{out} : \mathcal{A} \to \wp(\mathcal{L})$	Returns the set of all outgoing links of an activity.
$\mathcal{V}$	Set of all variables
$\mathsf{jc} : \mathcal{A} \to \mathcal{C} \cup \{\bot\}$	Returns the join condition of the given activity. "$\bot$" denotes an undefined join condition defaulting to the logical OR of all incoming links.
$\mathcal{L} : \mathcal{C} \cup \{\bot\} \to \wp(\mathcal{L})$	If $\mathcal{L}$ used as function, it returns the set of used links in a condition.
$\mathsf{tc}_{ex} : \mathcal{L} \to \mathbb{B}$	Returns `true` iff there is a transition condition on the given link.

Table 1: Notations used

Notation	Meaning
$\mathcal{B} = \mathcal{E} \cup \mathcal{C} \cup \{\bot\}$	The set of connecting labels
$\mathcal{E}$	The set of all events
$\mathcal{T}_\mathcal{E} = \{fault, compensation\}$	The set of all event types
$\mathsf{type}_\mathcal{E} : \mathcal{E} \to \mathcal{T}_\mathcal{E}$	Returns the type of the given event
$\mathcal{E}_{compensation}$	The set of all events e, where $\mathsf{type}_\mathcal{E}(e) = compensation$
$\mathcal{E}_{fault}$	The set of all events e, where $\mathsf{type}_\mathcal{E}(e) = fault$
$\mathsf{children} : \mathcal{A} \to \wp(\mathcal{A})$	Returns the set of all children of the given activity with respect to the hierarchy relation.
$\mathsf{descendants} : \mathcal{A} \to \wp(\mathcal{A})$	Returns the set of all transitive children of the given activity with respect to the hierarchy relation.
$\mathsf{clan} : \mathcal{A} \to \wp(\mathcal{A})$	Returns the set of all transitive children of the given activity and the given activity itself with respect to the hierarchy relation.
$\mathcal{A}_\mathcal{H}^{compensation} : \mathcal{A}_{scope} \to \wp(\mathcal{A})$	Returns the set of all activities used to handle the compensation event of the given scope. $s \mapsto \{a \mid \exists (s, e, a_{ch}), e \in \mathcal{E}_{compensation}, a \in \mathsf{clan}(a_{ch})\}$

Table 2: Details of the hierarchy relation HR

4.1 Handling Complex Types

Variables in BPEL processes are accessed using queries. Let $\mathcal{Q}_\mathcal{V}$ denote the set of all queries for accessing locations of variables. In the context of XPath [W3C99], such a query is a location path. If the whole variable is accessed, the query is empty (ϵ). We define $\mathcal{V}^{\mathcal{Q}_\mathcal{V}} \subseteq \mathcal{V} \times \mathcal{Q}_\mathcal{V}$ to denote all tuples of variables and valid queries on each variable. In the subsequent sections, each element in $\mathcal{V}^{\mathcal{Q}_\mathcal{V}}$ is called *variable element* to ease reading. A variable element is put into $\mathcal{V}^{\mathcal{Q}_\mathcal{V}}$ if an activity in the analyzed process reads or writes to that variable element.

Let `$var` be a variable of a complex type and thus being of the form presented in Figure 2. Assume now three writers w_1, w_2, w_3 in a sequence with following writes: w_1: `$var`, w_2: `$var/car`, w_3: `$var/car`. w_3 overwrites the data written by w_2, but it does not overwrite all data written by w_1. Thus, w_1 is still a writer to take into account for a subsequent read on `$var`, where the data written by w_1 and w_2 has to be merged.

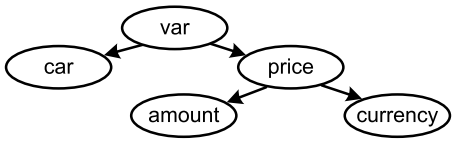

Figure 2: Variable of a complex type

Notation	Meaning
$\mathcal{V}$	Set of all variables including the variable implictly declared at fault handlers
$\mathsf{faultVariable}_{\mathcal{E}} : \mathcal{E}_{fault} \to \mathcal{V}$	Is the function which returns the variable declared by the given fault event. The result of the function corresponds to the content of the `faultVariable` element in a `catch` block of a fault handler.
$\mathcal{Q}_{\mathcal{V}}$	Set of all queries (i.e., XPath location paths) for accessing locations in variables. $\mathcal{Q}_{\mathcal{V}}$ includes 'ϵ' to denote "empty query", i.e., the whole variable is accessed
$\mathcal{V}^{\mathcal{Q}_{\mathcal{V}}} \subseteq \mathcal{V} \times \mathcal{Q}_{\mathcal{V}}$	The set of tuples of a variable v and a (valid) query q on v. Every element in $\mathcal{V}^{\mathcal{Q}_{\mathcal{V}}}$ is called "variable element" ease reading.
$\mathcal{V}_r^{\mathcal{Q}_{\mathcal{V}}} \subseteq \mathcal{V}^{\mathcal{Q}_{\mathcal{V}}}$	Is the set of variable elements read by reading activities
$\mathcal{V}_w^{\mathcal{Q}_{\mathcal{V}}} \subseteq \mathcal{V}^{\mathcal{Q}_{\mathcal{V}}}$	Is the set of variable elements written by writing activities
$w : \mathcal{A} \times \mathcal{V}^{\mathcal{Q}_{\mathcal{V}}} \to \mathbb{B}$	Returns `true` iff the given activity completely changes the given variable element. Cf. section 4.1
$r : \mathcal{A} \cup \mathcal{L} \times \mathcal{V}^{\mathcal{Q}_{\mathcal{V}}} \to \mathbb{B}$	Returns `true` iff the given activity or link completely reads the given variable element.

Table 3: Notation related to variables

A writer has to be removed from the set of possible writers if its written data is overwritten by a subsequent write (cf. section 4.4 on page 13). 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 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 location paths 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 location paths not to contain any variable reference and to return exactly one element in a variable. In consequence, the location path does not contain any variable reference, and returns an *lvalue*.

We distinguish between the variable elements addressed at readers and writers. $\mathcal{V}_r^{\mathcal{Q}_\mathcal{V}}$ is the set of variable elements addressed at reading activities and $\mathcal{V}_w^{\mathcal{Q}_\mathcal{V}}$ is the set of variable elements addressed at writing activities.

Assume $\mathsf{poss}_\circ : (\mathcal{A} \cup \mathcal{L}) \times \mathcal{V}^{\mathcal{Q}_\mathcal{V}} \to \wp(\mathcal{A})$ being the function returning the set of possible writers for a given activity or link and variable element during program execution (cf. definition 1 on the next page). 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 $\mathcal{W}_{\not\parallel} : \mathcal{A} \cup \mathcal{L} \times \mathcal{V}^{\mathcal{Q}_\mathcal{V}} \to \wp(\mathcal{A})$ returning the set of possible writers, including partial writers, is defined as follows:

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

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

4.2 Abstract Interpretation of a BPEL Process

The BPEL process definition is abstractly interpreted, where the results of all possible executions are merged. The merge does not contain writes on parallel branches, but supports parallel execution. This means that the result of parallel branches is joined at a node where the branches join. However, writes on a branch are not considered as writes on parallel branches. For example, in the BPEL process presented in Figure 3 on the following page, $\mathsf{w_a}$ and $\mathsf{w_b}$ are considered as writers for $\mathsf{r_b}$, but $\mathsf{w_b}$ is the only writer for $\mathsf{r_a}$.

The result of the abstract interpretation can be used to determine $\mathcal{W}_\parallel$, where writes in parallel branches are also considered. Since the restrictions ensure that there may be no writes happening in parallel to reads on the same variable, $\mathcal{W}_\parallel$ does not need to be determined.

The idea of the abstract interpretation 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, $\mathsf{w_2}$ is a possible writer

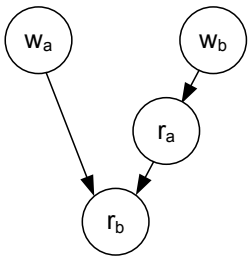

Figure 3: Parallel writes on a variable

for r_1 in the BPEL process presented in Figure 1 on page 4. w is a disabled writer if it is overwritten by a subsequent writer. For example, w_1 is a disabled by w_2. If the join condition jc_1 is an AND, w_1 will also remain disabled at r_1: If tc_1 evaluates to `false`, w_2 is dead. Thus, the status of l_1 is `false` and jc_1 subsequently evaluates to `false`. Thus, r_1 is dead and will not read the value written by w_1. A disabled writer can be a possible writer again, which happens for w_1 at r_2, if jc_1 is and AND. Since l_4 does not contain a transition condition, the status of l_4 is `true` and the join condition at r_2 evaluates to `true` even if the status of l_3 is `false`. Thus, the data written by w_1 does not reach r_1, but r_2. A writer can also be disabled completely and thus become invalid. Assume tc_1 is `true`. Then w_2 always overwrites the value written by w_1 and thus w_1 is invalid at all activities following w_2.

To store the state of a writing activity, three lattices and a Boolean value are used: One lattice for the possible writers, one lattice for the disabled writers, one lattice for the invalid writers, and one Boolean value for marking the current activity to be possibly dead. 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.

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

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

Definition 1: $\mathsf{poss}_\circ(x, v_e)$
$\mathsf{poss}_\circ(x, v_e) := \pi_1(\mathsf{writes}_\circ(x, v_e))$ *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".*

Definition 2: $\mathsf{dis}_\circ(x, v_e)$
$\mathsf{dis}_\circ(x, v_e) := \pi_2(\mathsf{writes}_\circ(x, v_e))$ *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.

Definition 3: $\mathsf{inv}_\circ(x, v_e)$
$\mathsf{inv}_\circ(x, v_e) := \pi_3(\mathsf{writes}_\circ(x, v_e))$ *returns the activities which can never be revived at position* x. *"inv" stands for "invalid".*

The writers contained in $\mathsf{inv}_\circ$ are needed for an efficient construction of concurrent reads, which is future work.

Definition 4: $\mathsf{mbd}_\circ(x, v_e)$
$\mathsf{mbd}_\circ(x, v_e) := \pi_4(\mathsf{writes}_\circ(x, v_e))$ *returns* `true` *iff* x *can be disabled 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. "Directly preceding writer" is defined as follows:

Definition 5: Directly preceding writer
A writer w *is a directly preceding writer if there is no other writer on the path from* w *to* x.

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

4.3 Depth-first Search Covering All Possible Executions

The abstract interpretation executes a depth-first search (DFS) which covers all possible executions. Since the links in the program graph can contain a transition condition, the DFS visits the links explicitly. A DFS is started for every variable element $v_e \in \mathcal{V}_w^{\mathcal{Q}_\mathcal{V}}$ as shown in Algorithm 1. Each variable can be analyzed independently from the others, since BPEL does not support aliasing of variables. "Aliasing" describes that two variables can point to the same data place in memory. That means, if variable x is an alias of variable y, x:=1 also sets y to 1.

After AnalyzeProcessModel completed, $\mathsf{writes}_\circ$ and $\mathsf{writes}_\bullet$ are defined for all variables and all queries on every variable. In section 4.10 on page 20 we present how to use $\mathsf{writes}_\circ$ to derive the def-use edges of the corresponding BPEL process.

The DFS is mainly implemented in HandleActivity and HandleLink. Section 4.9 on page 20 deals with HandleLink that visits the control links. The activities are visited by HandleActivity. It calls HandleBasicActivity to handle basic activities, HandleFlow to handle the flow activity, HandleWhile to handle the while loop and finally HandleScope to handle a scope. HandleActivity itself checks whether all the incoming links of the currently visited activity were handled and the parent activity was visited. If not, it returns, since the activity will be reached via all non-visited links or

Algorithm 1 Analysis of a given BPEL process

procedure AnalyzeProcessModel(M)
  Determine $\mathcal{V}_w^{\mathcal{Qv}}$ in the given BPEL process model M.
  for all $v_e \in \mathcal{V}_w^{\mathcal{Qv}}$ **do**
    $\forall a \in \mathcal{A} : \mathsf{visited}(a) \leftarrow \texttt{false}$
    $\forall l \in \mathcal{L} : \mathsf{visited}(l) \leftarrow \texttt{false}$
    HandleActvity(process, v_e) // process is the process activity in M
  end for
end procedure

Algorithm 2 Handling of an activity

procedure HandleActivity(a,v_e)
  $\mathsf{parentHandled} \leftarrow (a = \mathsf{process}) \vee \mathsf{visited}(a'), (a', b, a) \in \mathsf{HR}$
    // $(a', b, a) \in \mathsf{HR}$ denotes that a' is the parent of a
  if $\mathsf{parentHandled} \wedge (\mathcal{L}_{in}(a) = \emptyset \vee \forall l \in \mathcal{L}_{in}(a) : \mathsf{visited}(l))$ **then**
    $\mathsf{visited}(a) \leftarrow \texttt{true}$
    $$\mathsf{writes}_\circ(a, v_e) \leftarrow \begin{cases} \mathsf{joinLinks}(a, v_e) & |\mathcal{L}_{in}(a)| > 0 \\ \mathsf{writes}_\circ(p, v_e) & \exists p : (p, c, a) \in \mathsf{HR} \\ (\emptyset, \emptyset, \emptyset, \texttt{false}) & \text{otherwise} \end{cases}$$
    if $a \in \mathcal{A}_{basic}$ **then**
      HandleBasicActivity(a,v_e)
    else if $a \in \mathcal{A}_{flow}$ **then**
      HandleFlow(a,v_e)
    else if $a \in \mathcal{A}_{scope}$ **then**
      HandleScope(a,v_e)
    else if $a \in \mathcal{A}_{while}$ **then**
      HandleWhile(a,v_e)
    else if $a = \mathsf{process}$ **then** // Directly handle the contained activity
      HandleActivity(a',v_e), $(a, \bot, a') \in \mathsf{HR}$
      $\mathsf{writes}_\bullet(a) \leftarrow \mathsf{writes}_\bullet(a')$
    end if
    for all $l \in \mathcal{L}_{out}(a)$ **do**
      HandleLink(l,v_e)
    end for
  end if
end procedure

via the parent activity again. If all incoming links and the parent activity were handled, the information of the predecessors of the activity have to be put into $\mathsf{writes}_\bullet$. If the activity has incoming links, their information has to be joined, since the execution may

have reached the activity by these links. 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., 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, which are presented in the subsequent sections. 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. 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.

4.4 Joining the Information of Incoming Links

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

Algorithm 3 Determining the set P of possible writers

if $|\mathcal{L}_{in}(a)| = 1 \vee$ the join condition of a is AND over all incoming links **then**

$$P \leftarrow \bigcup_{l \in \mathcal{L}_{in}(a)} \mathsf{poss}_\bullet(l, v_e) \tag{1}$$

else

$$P \leftarrow \bigcup_{l \in \mathcal{L}_{in}(a)} \mathsf{poss}_\bullet(l, v_e) \cup \mathsf{dis}_\bullet(l, v_e) \setminus \bigcap_{l \in \mathcal{L}_{in}(a)} \mathsf{dis}_\bullet(l, v_e) \tag{2}$$

end if

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. (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 revival may happen later as illustrated on activity r_2 in figure 1 on page 4. (ii) If there is only one incoming link, the current activity can only be reached over that path. 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.

Case (2) handles the case where the join can enable disabled writes. An active incoming link not on 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 r_2 in figure 1 on page 4, where w_1 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 disabled during the execution on all paths reaching the current activity. These writes cannot be enabled again by an incoming path and thus are not enabled again.

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 the 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 $|\mathcal{L}_{in}(a)| = 1 \vee$ the join condition of a is AND over all incoming links **then**

$$D \leftarrow \bigcup_{l \in \mathcal{L}_{in}(a)} \mathsf{dis}_\bullet(l, v_e) \tag{1}$$

else

$$D \leftarrow \bigcap_{l \in \mathcal{L}_{in}(a)} \mathsf{dis}_\bullet(l, v_e) \tag{2}$$

end if

The set I of invalid writers The invalid writes are collected and not modified, since they are used for the analysis of the concurrent writes only.

Algorithm 5 Determining the set I of invalid writers

$$I \leftarrow \bigcup_{l \in \mathcal{L}_{in}(a)} \mathsf{inv}_\bullet(l, v_e)$$

State "dead" of an activity An activity a may be disabled 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`. A disabled activity is also known as a dead activity.

The decision logic presented in section 4.5 on page 16 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. A writer marked as "invalid" will never be 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 marked 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`.

Therefore, 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.

For case i we define the function $\mathsf{alwaystrue} : \mathcal{C} \cup \{\bot\} \to \mathbb{B}$ to return `true` iff the given join condition always evaluates to `true`. The function may have a one-sided error: If the join condition always evaluates to `true`, alwaystrue may return `false`, but not the other way round.

For case ii, we define the function $\mathsf{negations} : \mathcal{C} \cup \{\bot\} \to \mathbb{B}$ 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.

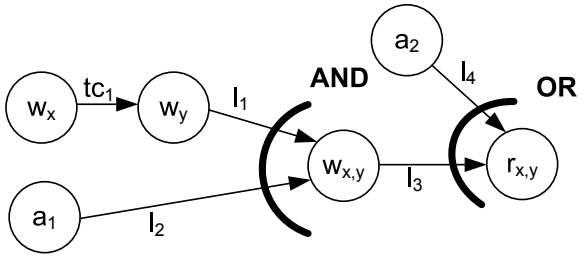

Figure 4: A process illustrating the usage of $[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$

We illustrate the use of $[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$ by the process presented in figure 4, which is a modified version of the process presented in figure 1 on page 4. $\mathsf{w_x}$ and $\mathsf{w_y}$ write to different variables x and y. Furthermore $\mathsf{w_{x,y}}$ writes to both x and y. $\mathsf{r_{x,y}}$ is reading from

both x and y. $\mathsf{l_i}$ are links without transition conditions. $\mathsf{w_x}$ and $\mathsf{w_{x,y}}$ both write to x. Because of the transition condition $\mathsf{tc_1}$ and the AND join on $\mathsf{w_{x,y}}$, $\mathsf{w_{x,y}}$ can be disabled and the value of $\mathsf{w_x}$ reach $\mathsf{r_{x,y}}$. On the other hand, the value written by $\mathsf{w_y}$ never reaches $\mathsf{r_{x,y}}$. If $\mathsf{w_y}$ is disabled, $\mathsf{w_{x,y}}$ is disabled, too. If $\mathsf{w_y}$ is not disabled, $\mathsf{w_{x,y}}$ is not disabled 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 $\mathsf{mbd}_\bullet(l, v_e) = \texttt{true}$. Recall that $\mathsf{mbd}_\bullet(l, v_e) = \texttt{true}$ denotes that there exists a path from the root node to the current node, which disables link l. That disablement may happen in some cases, but not in all. In a process execution, the activity b is executed iff the status of the incoming link l evaluates to `false`.

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

The negated value of $[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$ is used in Algorithm 4.4 to determine d. In our case, $\neg[\![b, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet} = \neg\texttt{true} = \texttt{false}$ 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`. That means, that $[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$ cannot be used if the join condition of an activity contains negations.

Algorithm 4.4 presents the determination of d.

Algorithm 6 Determining whether an activity may be dead

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

4.5 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 $\mathsf{writes}_\bullet(a, v_e)$, an abstract interpretation of the activity is done regarding whether it writes to the given variable element. If a is not a writing activity (e.g., empty or while activity), $\mathsf{writes}_\bullet(a, v_e)$ is the identity of $\mathsf{writes}_\circ(a, v_e)$ (Case (1) in Algorithm 7 shown below).

If a is a writing activity, the result depends on the value of $\mathsf{mbd}_\circ(a, v_e)$. If $\mathsf{mbd}_\circ(a, v_e) = \texttt{true}$ (i.e., a can be disabled on a path from any directly preceding writer, or from the

root node if there is no directly preceding writer, to a), the possible writes are added to the disabled writes and a is put as the only writer (Case (2) in Algorithm 7). 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 executed. If w was executed, a may be not be executed ($\mathsf{mbd}_\circ(a, v_e) = \texttt{true}$). Therefore, w has to be stored as w can become valid again at a successor of a.

In the other case, where $\mathsf{mbd}_\circ(a, v_e) = \texttt{false}$, the possible writes can never be active again and become invalid (Case (3) in Algorithm 7). 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. The write of w is always overwritten by the write of a in this case.

If a is a writer to v_e, $\mathsf{mbd}_\bullet(a, v_e)$ is set to `false` in Algorithm 7. In this case, a is the starting point of all paths from a to a subsequent writer. a is not disabled at the beginning of each path starting at a. Thus, $\mathsf{mbd}_\bullet$ has to be `false`. On the other hand, Algorithm 6 on the preceding page and Algorithm 11 on page 20 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.

Algorithm 7 Handling a basic activity

procedure HANDLEBASICACTIVITY(a,v_e)

$\mathsf{writes}_\bullet(a, v_e) \leftarrow$

$$\begin{cases} \mathsf{writes}_\circ(a, v_e) & \neg w(a) & (1) \\ (\{a\}, \mathsf{dis}_\circ(a, v_e) \cup \mathsf{poss}_\circ(a, v_e), \mathsf{inv}_\circ(a, v_e), \texttt{false}) & w(a, v_e) \wedge \mathsf{mbd}_\circ(a, v_e) & (2) \\ (\{a\}, \mathsf{dis}_\circ(a, v_e), \mathsf{inv}_\circ(a, v_e) \cup \mathsf{pos}_\circ(a, v_e), \texttt{false}) & w(a, v_e) \wedge \neg\mathsf{mbd}_\circ(a, v_e) & (3) \end{cases}$$

end procedure

4.6 Handling a Flow Activity

Algorithm 8 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 $r \in roots$ **do**

HANDLEACTIVITY(r,v_e)

end for

$\mathsf{writes}_\bullet(f, v_e) \leftarrow$

$(\bigcup a \in \mathsf{poss}_\bullet(a', v_e), \bigcup a \in \mathsf{dis}_\bullet(a', v_e), \bigcup a \in \mathsf{inv}_\bullet(a', v_e), \bigvee a \in \mathsf{mbd}_\bullet(a', v_e))$,
$a' \in \{a \mid a \in \mathsf{children}(f) \wedge \neg\exists l : (a, l, a'') \in \mathsf{LR} : a'' \in \mathsf{descendants}(f)\}$

end procedure

The traversal of the activities nested in a flow activity starts from the roots of the flow activity. 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.

4.7 Handling While Activities

Algorithm 9 Handling of while activities

procedure HandleWhileActivity(w,v_e)
  Let n be the activity nested in the while loop. $n : (w, c, n) \in \mathsf{HR}$
  repeat
    $last \leftarrow \mathsf{writes}_\circ(w, v_e)$
    HandleActivity(n,v_e)
    $$\mathsf{writes}_\circ(w, v_e) \leftarrow \begin{pmatrix} \pi_1(last) \cup \pi_1(\mathsf{writes}_\bullet(n, v_e)), \\ \pi_2(last) \cup \pi_2(\mathsf{writes}_\bullet(n, v_e)), \\ \pi_3(last) \cup \pi_3(\mathsf{writes}_\bullet(n, v_e)), \\ \pi_4(last) \vee \pi_4(\mathsf{writes}_\bullet(n, v_e)) \end{pmatrix}$$
  until $last = \mathsf{writes}_\circ(w, v)$
  $\mathsf{writes}_\bullet(w, v) \leftarrow \mathsf{writes}_\circ(w, v)$
end procedure

In while activities, $\mathsf{writes}_\circ(w, v_e)$ is not only joining the information of the incoming links, but also takes into account the information produced by the nested activity. This information is joined with the current information of $\mathsf{writes}_\circ(w, v_e)$ until the previous value of $\mathsf{writes}_\circ(w, v_e)$ equals the current value, i.e., a fix point is reached. Termination is guaranteed, since $\wp(\mathcal{A})$ is a finite set and there are only elements added to the set in HandleActivity. This abstract interpretation of the while activity is valid, since BPEL also first executes the while activity until the loop condition is `false` and visits the outgoing links afterwards. Note that the while activity itself does not write to any variable. Thus the final value of $\mathsf{writes}_\circ$ can be directly copied to $\mathsf{writes}_\bullet$.

4.8 Handling Scope Activities

A scope first executes the nested activity. If the execution fails, a fault handler is triggered. Since the restrictions state that the data to the fault handler is sent by the coordinator, the only data which can reach the activities nested in the fault handler is the data contained in the `faultVariable`. Since no sending/receiving blocks are created for the `faultVariable` [Kha07], we are ignoring writers to the `faultVariable` and writers

$$\mathsf{writes}_\circ(a, v_e) \leftarrow \begin{cases} \mathsf{joinLinks}(a, v_e) & |\mathcal{L}_{in}(a)| > 0 \\ (\emptyset, \emptyset, \emptyset, \texttt{false}) & \exists (p, e, a) \in \mathsf{HR}, \mathsf{type}_{\mathcal{E}}(e) = \mathit{fault} \\ \mathsf{writes}_\bullet(p, v_e) & \exists (p, e, a) \in \mathsf{HR}, \mathsf{type}_{\mathcal{E}}(e) = \mathit{compensation} \\ \mathsf{writes}_\circ(p, v_e) & \exists p : (p, c, a) \in \mathsf{HR} \\ (\emptyset, \emptyset, \emptyset, \texttt{false}) & \text{otherwise} \end{cases}$$

Figure 5: Modified $\mathsf{writes}_\circ$ to handle the fault handling and compensation activity

not included in the fault handler. To support this, HandleActivity has to be adapted to reset $\mathsf{writes}_\circ$. The modified assignment to $\mathsf{writes}_\circ$ is shown in figure 5. After the fault handler runs, the data is available for the activities following the scope where the fault handler is declared. Thus, $\mathsf{writes}_\bullet(s, v_e)$ has to include the data written by each fault handler *fh*. The inclusion of the results is done via the $\sqcup$ operator, denoting the union of each set in $\mathsf{writes}_\bullet$ and the disjunction of $\mathsf{mbd}_\bullet$:

$$(a, b, c, d) \sqcup (a', b', c', d') := (a \cup a', b \cup b', c \cup c', d \vee d')$$

After the activity nested in the scope activity is finished, the compensation handler will get a snapshot of the variable values in BPEL 1.1. Thus, the initialized value of the compensation handler is $\mathsf{writes}_\bullet(s, v_e)$ as shown in figure 5. If the compensation handler is executed, it works on local copies of the variables. Thus, data written by compensation handlers never crosses the border of the compensation handler and $\mathsf{writes}_\bullet$ of the compensation handler may be ignored.

Algorithm 10 presents the complete handling of a scope activity.

Algorithm 10 Analysis of a scope activity

procedure HandleScope(a,v_e)
  HandleActivity(a',v_e), $(a, \bot, a') \in \mathsf{HR}$
  $\mathsf{writes}_\bullet(a, v_e) \leftarrow \mathsf{writes}_\bullet(a', v_e)$
  $FH \leftarrow \{fh \mid (a, e, fh) \in \mathsf{HR}, \mathsf{type}_{\mathcal{E}}(e) = \mathit{fault}\}$
  for all $fh \in FH$ **do**
    HandleActivity(fh,v_e)
  end for
  if $\exists (a, e, ch) \in \mathsf{HR} : \mathsf{type}_{\mathcal{E}}(e) = \mathit{compensation}$ **then**
    HandleActivity(ch,v_e)
  end if
  for all $fh \in FH$ **do**
    $\mathsf{writes}_\bullet(a, v_e) \leftarrow \mathsf{writes}_\bullet(a, v_e) \sqcup \mathsf{writes}_\bullet(fh, v_e)$
  end for
end procedure

4.9 Handling Links

A control link has exactly one source and 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 may contain a transition condition, thus turning the subsequent activity to dead. The subsequent activity may be dead, if the source of the link may be dead or if there is a transition condition on link l.

A special case is the case when a transition condition always evaluates to `true`. In this case, the value of $\mathsf{mbd}_\circ$ does not change, 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 [BFG05]. We define the function $\mathsf{alwaysTrue}$ to return `true`, if the given transition condition always evaluates to `true` and `false` in all other cases. This includes, that $\mathsf{alwaysTrue}$ returns `false` for undecidable cases. Thus, the algorithm produces an over-approximation, as illustrated in section 4.4 on page 15.

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 11.

Algorithm 11 Handling a link

procedure HANDLELINK(l,v_e)
  $a, a' : (a, l, a') \in \mathsf{HR}$
  $\mathsf{writes}_\circ(l, v_e) \leftarrow \mathsf{writes}_\bullet(a, v_e)$
  $\mathsf{writes}_\bullet(l, v_e) \leftarrow \begin{pmatrix} \pi_1(\mathsf{writes}_\circ(l, v_e)), \pi_2(\mathsf{writes}_\circ(l, v_e)), \pi_3(\mathsf{writes}_\circ(l, v_e)), \\ \pi_4(\mathsf{writes}_\circ(l, v_e)) \vee (\mathsf{tc}_{ex}(l) \wedge \neg\mathsf{alwaysTrue}(\mathsf{tc}(l))) \end{pmatrix}$
  $\mathsf{visited}(l) \leftarrow \mathtt{true}$
  HANDLEACTIVITY(a',v_e)
end procedure

4.10 Creating Def-Use Edges

Once the process model has been analyzed, $\mathsf{poss}_\circ(a, v_e)$ returns the set of possible writers for every given activity and variable element. As shown in section 4.1, a read of an activity a on a variable element v_e in a lattice ($r(a, v_e) = \mathtt{true}$) also has to read all data written to children of v_e. Thus, the function $\mathsf{CE} : \mathcal{A} \cup \mathcal{L} \to \mathcal{A} \times \mathcal{A}$ returning the def-use edges for an activity or a link is defined as follows:

$$\mathsf{CE} : x \mapsto \{(a', x) \mid a' \in \mathsf{poss}_\circ(a, v'_e),\ v'_e \sqsubseteq v_e,\ r(a, v_e) = \mathtt{true}\}$$

The order in which the sources of the links complete has to be regarded at the target activity. Otherwise stale data may overwrite fresh data written by a succeeding writer. The rule 'last writer to successfully complete wins' can be used to resolve write conflicts.

Upon splitting a process and distributing the fragments onto different network locations, however, messages may get reordered on the network and there is no guarantee of timing. In [KKL07], we combine the output of this algorithm along with control dependencies between the writers to provide a solution for resolving conflicts across process fragments. Since [KKL07] does not directly use def-use edges, but a special function Q_s, we present in the next section how to create Q_s out of the result of the data-flow algorithm.

5 Creating Q_s

The algorithm presented in [KKL07] demands a function $Q_s : \mathcal{A} \times \mathcal{V} \to \wp(\wp(\mathcal{Q}_\mathcal{V}) \times \wp(\mathcal{A}))$.

Definition 6: The function Q_s
The function Q_s takes as argument an activity and a variable and returns a set of tuples of queries and writers to all of these queries. The writers returned write to variable elements read by the given activity or a transition condition of a link leaving the activity.

In other words, $Q_s(a, v)$ groups sets of queries on v with writers which may have written to the same parts of x expressed in those queries by the time a is reached in the control flow. Thus, $Q_s(a, v)$ returns a set of tuples, each containing a query set and a writer set. Consider w_1, w_2 and w_3 that write to v such that their writes are visible to a when a is reached. Assume they respectively write to $\{v.b, v.c\}$, $\{v.b, v.c, v.d\}$ and $\{v.d, v.e\}$. Then the result of $Q_s(a, v)$ is as follows:

$$\begin{aligned} Q_s(a, v) = \{&(\{.b, .c\}, \{\mathsf{w}_1, \mathsf{w}_2\}),\\ &(\{.d\}, \{\mathsf{w}_2, \mathsf{w}_3\}),\\ &(\{.e\}, \{\mathsf{w}_3\})\} \end{aligned}$$

A property of an element $(Q, A) \in Q_s(a, v)$ is that all $w \in A$ have written to each $q \in Q$. Thus, there may be a $(Q', A') \in Q_s(a, v)$, $(Q', A') \neq (Q, A)$, such that $A' \cap A \neq \emptyset$. In the example, this is the case for w_2 and w_3, since the set of queries they write to are not equal, but overlap. The difference between $\mathsf{poss}_\circ(a, v)$ (defined at definition 1 on page 10) and $Q_s(a, v)$ is that the data of a writer contained in Q_s is read at activity a, whereas the data written by a writer in $\mathsf{poss}_\circ(a, v)$ may not necessarily be read by a.

[KKL07] excluded loops and scopes. However, the work of [KL06] is extended in [Kha07] to support splitting of loops and scopes of BPEL-D processes. If loops and scopes should be treated in the algorithm in [KKL07], certain writers have to be merged or excluded from Q_s. Therefore, we first present the construction of the helper function $Q_s^* : \mathcal{A} \times \mathcal{V} \to \wp(\wp(\mathcal{Q}_\mathcal{V}) \times \wp(\mathcal{A}))$ out of the result of the data-flow analysis. If scopes and loops are not regarded, Q_s^* and Q_s are the same functions. Otherwise, the result of Q_s^* has to be modified to return a Q_s which can be treated by the algorithm presented in [KKL07]. In section 5.3, we show how Q_s^* is used by the function Q_s to return the required result. In the following, we present the construction of Q_s^*.

Note that a function $f : A \to B$ can also be written as a set of tuples $f \subset A \times B$. Each element in the range of Q_s is of the form $\wp(\mathcal{Q}_\mathcal{V}) \times \wp(\mathcal{A})$. According to definition 6, this tuple set is a function assigning queries to their writer. We use this property to construct Q_s^*: We define a function $q : \wp(\mathcal{Q}_\mathcal{V}) \to \wp(\mathcal{A})$ and stepwise add or modify it with mappings from $\wp(\mathcal{Q}_\mathcal{V})$ to $\wp(\mathcal{A})$.

The construction of Q_s^* uses the function $Q_w : \mathcal{A} \times \mathcal{A} \times \mathcal{V} \to \wp(\mathcal{Q}_\mathcal{V})$, which returns the set of queries on the given variable that are written by the first activity and read at the second activity or at transition conditions of the second activity. As presented in section 4.1, a write to a variable element v_e is also a write to every $v'_e \sqsubset v_e$. We therefore remove each v'_e, where $v'_e \sqsubset v_e$ from the result, if v_e is contained in the result (line 4 in Algorithm 12).

Algorithm 12 The function Q_w returning the set of queries on the given variable that are written by the first activity and read at the second activity or at transition conditions of links of the second activity

1: **function** $Q_w(a_w,a_r,v)$
$R \leftarrow \{q \in \mathcal{Q}_\mathcal{V} \mid \exists v'_e = (v,q) \exists v_e \sqsupseteq v'_e :$
2: $(r(a_r, v_e) \wedge a_w \in \mathsf{poss}_\circ(a_r, v'_e)) \vee$
$(\exists l \in \mathcal{L} : (a_r, l, a') \in \mathsf{LR} \wedge r(l, v_e) = \mathtt{true} \wedge a_w \in \mathsf{poss}_\circ(l, v'_e))\}$
3: // Remove all obsolete queries
4: **while** $\exists q, q' \in R : q \sqsubset q'$ **do**
5: $R \leftarrow R \setminus \{q\}$
6: **end while**
7: **return** R
8: **end function**

Algorithm 13 on the following page presents the construction of Q_s^*. Since Q_s^* is defined on an activity and a variable, the algorithm iterates over all activities (a_r, r stands for "reading", line 3) and variables (v, line 4). Since an activity also reads the variable elements needed by its outgoing links, the set PW is built to contain both the possible writers for the activity and its outgoing links (line 6). In line 7, the activity itself is removed from PW, since the value written by the activity (and read by its outgoing links) is available at its links. Lines 8 to 16 iterate over all writers and check whether there already is a set of writers for the set of queries the current writer (a_w) is writing to (Q, line 10). If not, the current writer is put as the only writer for Q in q (line 14), otherwise the result of q is adopted for Q (line 11f.[2]). After the loop ran, q is defined for all queries and writers on the queries for the reading activity a_r and variable v. Since q is a set of tuples with the same type as $\pi_3(Q_s^*)$ ($q \subset \wp(\mathcal{Q}_\mathcal{V}) \times \wp(\mathcal{A})$), it can be directly used as $\pi_3(Q_s^*)$ when adding the result for a_r and v to Q_s^* (line 17). Here, it is important to note that a function $g : C \times D \to E$ may also be written as $g \subset C \times D \times E$.

[2] "f." stands for "and the subsequent line"

Algorithm 13 Iterative construction of Q_s^*

1: **procedure** GENERATEQ_s^*
2: $Q_s^* \leftarrow \emptyset$
3: **for all** $v \in \mathcal{V}$ **do**
4: **for all** $a_r \in \mathcal{A}$ **do**
5: $q \leftarrow \emptyset$
6: $PW \leftarrow \bigcup_{\left\{ v_e \,\middle|\, v_e \in \mathcal{V}_w^{\mathcal{Q}\mathcal{V}},\, v_e = (v,q) \right\}} \left(\mathsf{poss}_\circ(a_r, v_e) \cup \bigcup_{\{l \,|\, (a_r,l,a') \in \mathsf{LR}\}} \mathsf{poss}_\circ(l, v_e) \right)$
7: $PW \leftarrow PW \setminus \{a_r\}$
8: **for all** $a_w \in PW$ **do**
9: $Q \leftarrow Q_w(a_w, a_r, v)$
10: **if** $\exists (Q, W') \in q$ **then**
11: $W \leftarrow W' \cup \{a_w\}$
12: $q \leftarrow q \setminus \{(Q, W')\} \cup \{(Q, W)\}$
13: **else**
14: $q \leftarrow q \cup \{(Q, \{a_w\})\}$
15: **end if**
16: **end for**
17: $Q_s^* \leftarrow Q_s^* \cup \{(a_r, v, q)\}$
18: **end for**
19: **end for**
20: **end procedure**

5.1 Treatment of Loops

For creating receiving flows for a loop l, the following cases have to be distinguished:

1. Data reaching the loop before the first iteration
2. Data produced in the current iteration reaching a reading activity in the loop
3. Data produced in the previous iteration reaching a reading activity in the loop
4. Data produced for activities following the loop

In case 1, the data is collected by a block before the loop. In case 2, the result of Q_s has only to contain those writers. The constructions for the other cases ensure that data produced by other writers also reaches the reader. In case 3 the data is collected at the end of the loop. In case 4, the data is collected by one fragment inside the loop and sent after the loop completed to the readers. We call that sender for the data a *collecting writer* w_l. w_l is added to the set of activities as soon as it is created. Since w_l is used as a writer for subsequent reads, we define the function $\mathsf{w}_{\sum} : \mathcal{A}_{while} \to \mathcal{A}$ to return w_l for the given loop l.

To define the functions determining the required sets, we define the following functions: path, $\mathsf{path}_{\not\emptyset}$ and l. $\mathsf{path} : \mathcal{A} \times \mathcal{A} \to \mathbb{B}$ returns `true` iff there is a path from the first given activity to the second given activity. To ease the handling of compensation handlers, the only path to a compensation handler is from the end of the scope to its compensation handler. This definition ensures that the data for compensation handlers is correctly collected by the collecting writers. $\mathsf{path}_{\not\emptyset} : \mathcal{A} \times \mathcal{A} \to \mathbb{B}$ returns `true` iff there is a path from the first given activity to the second given activity ($\mathsf{path}(a, b) = \texttt{true}$) without considering paths introduced by looping. The formal definition of path and $\mathsf{path}_{\not\emptyset}$ is out of scope of this paper.

The function $\mathsf{l} : \mathcal{A} \to \mathcal{A}_{while} \cup \{\bot\}$ returns the directly enclosing loop for the current activity:

$$\mathsf{l} : a \mapsto \begin{cases} l & \exists l \in \mathcal{A}_{while}, a \in \mathsf{descendants}(l) \wedge \\ & \nexists\, l' : l' \in \mathcal{A}_{while}, a \in \mathsf{descendants}(l'), |\mathsf{descendants}(l')| < |\mathsf{descendants}(l)| \\ \bot & \text{otherwise} \end{cases}$$

Case 1: Data reaching the loop before the first iteration Data written before the loop and needed inside the loop or at the loop condition is collected before the loop. The function $Q_s^{preloop} : \mathcal{A}_{while} \times \mathcal{V} \to \wp(\wp(\mathcal{Q}_{\mathcal{V}}) \times \wp(\mathcal{A}))$ returns a set of tuples of queries and writers to all of these queries for the given loop and variable. The writers returned are located before the loop. All the $Q_s^*(a, v)$ of each activity a contained in the currently regarded loop l have to be merged into one $Q_s^{preloop\prime}(l, v)$ first. Since the merging is also needed for the compensation handler, we define $Q_s^{*,\cup} : \mathcal{A} \times \mathcal{V} \to \mathcal{A} \times \mathcal{V} \times \wp(\wp(\mathcal{Q}_{\mathcal{V}}) \times \wp(\mathcal{A}))$ to union the Q_s^* for the activities contained in the given activity and for the given variable.

Algorithm 14 The function $Q_s^{*,\cup}$, which returns the union of $Q_s^*(a, v)$ for activities contained in the given activity

```
1: function Q_s^{*,∪}(a,v)
2:     R ← ∅                                   // R : ℘(℘(Q_V) × ℘(A))
3:     for all a' ∈ clan(a) do
4:         (Q, W') ← Q_s^*(a', v)
5:         if ∃(Q, W) ∈ R then
6:             R ← R \ {(Q, W)} ∪ {(Q, W ∪ W')}
7:         else
8:             R ← R ∪ {(Q, W')}
9:         end if
10:    end for
11:    return R
12: end function
```

The implementation is presented in Algorithm 14. The merging is used in line 2 in Algorithm 15 on the next page, which implements $Q_s^{preloop}$. After the merging, all writers in l have to be taken out from the result (line 5). Let l_p be the parent loop of the l currently regarded. Writers outside of l_p have already been treated by the handling of l_p and may not be included in the result (line 7). Writers not preceding l have already been treated by the handling of l_p, too. Thus, these writers get removed, too (line 9). Lines 10 to 12 deal with the collecting writers created for case 4. If l_p contains other loops l_i, then the writers in l_i have to be combined to w_{l_i}, which has to be used instead of the writers nested in l_i. If no l_p exists, there is no treatment of the writers outside of l_p, but the writers contained in preceding loops have still to be combined.

Case 2: Data produced in the current iteration reaching a reading activity in the loop In this case, only the writers contained in the current loop l and preceding the given activity have to be returned by Q_s. Algorithm 16 on the following page presents the determination of the result of $Q_s(a, v)$ if a is directly nested in a loop. Lines 4 and 5 ensure that W only contains writers preceding the current activity. Lines 6ff.[3] ensure that collecting writers are used where appropriate.

Case 3: Data produced in the previous iteration reaching a reading activity in the loop In this case, the data is collected at the end of the loop. The function $Q_s^{intraloop} : \mathcal{A}_{while} \times \mathcal{V} \to \wp(\wp(\mathcal{Q}_\mathcal{V}) \times \wp(\mathcal{A}))$ returns a set of tuples of queries and writers to all of these queries for the given loop and variable required for the next loop iteration. Algorithm 17 on page 27 presents the implementation. Lines 5f. ensure that W only contains writers reaching a by looping. Lines 7ff. ensure that collecting writers are used

[3] "ff." (et folii) stands for "and the subsequent lines".

Algorithm 15 The function $Q_s^{preloop}$, which returns $Q_s(l, v)$ for a new reader collecting the data before the loop l starts

1: **function** $Q_s^{preloop}(l,v)$
2: $\quad Q_s^{preloop\prime} \leftarrow Q_s^{*,\cup}(l, v)$
3: $\quad R \leftarrow \emptyset$
4: $\quad$ **for all** $(Q, W') \in Q_s^{preloop\prime}$ **do**
5: $\quad\quad W' \leftarrow W' \setminus \mathsf{clan}(l)$
6: $\quad\quad$ **if** $\mathsf{l}(l) \neq \bot$ **then**
7: $\quad\quad\quad W' \leftarrow W' \cap \mathsf{clan}(\mathsf{l}(l))$
8: $\quad\quad$ **end if**
9: $\quad\quad W \leftarrow \{w \mid w \in W', \mathsf{path}_{\not\circlearrowleft}(w, l) = \mathtt{true}\}$
10: $\quad\quad$ **while** $\exists w \in W : \mathsf{l}(l) \neq \mathsf{l}(w)$ **do**
11: $\quad\quad\quad W \leftarrow W \setminus \{w\} \cup \{\mathsf{w}_{\sum}(w)\}$
12: $\quad\quad$ **end while**
13: $\quad\quad R \leftarrow R \cup \{(Q, W)\}$
14: $\quad$ **end for**
15: $\quad$ **return** R
16: **end function**

Algorithm 16 The function $Q_s^{\circlearrowleft}$ to determine Q_s for activities nested in loops

1: **function** $Q_s^{\circlearrowleft}(a,v)$
2: $\quad R \leftarrow \emptyset$
3: $\quad$ **for all** $(Q, W') \in Q_s^*(a, v)$ **do**
4: $\quad\quad W' \leftarrow W' \cap \mathsf{clan}(\mathsf{l}(a))$
5: $\quad\quad W \leftarrow \{w \mid w \in W', \mathsf{path}_{\not\circlearrowleft}(w, a) = \mathtt{true}\}$
6: $\quad\quad$ **while** $\exists w \in W : \mathsf{l}(w) \neq \mathsf{l}(a)$ **do**
7: $\quad\quad\quad W \leftarrow W \setminus \{w\} \cup \mathsf{w}_{\sum}(w)$
8: $\quad\quad$ **end while**
9: $\quad\quad$ **if** $W \neq \emptyset$ **then**
10: $\quad\quad\quad R \leftarrow R \cup \{(Q, W)\}$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: **end function**

where appropriate.

Algorithm 17 The function $Q_s^{intraloop}$, which determines $Q_s(l, v)$ for a new reader collecting the data at the end of the loop l for the next iteration

```
1:  function Q_s^intraloop(l,v)
2:      R ← ∅
3:      for all a ∈ descendants(l) do
4:          (Q, W) ← Q_s^*(a, v)
5:          W ← W ∩ clan(l)
6:          W ← W \ {a' | path_∅(a', a) = true}
7:          while ∃w ∈ W : l(w) ≠ l do
8:              W ← W \ {w} ∪ w_Σ(w)
9:          end while
10:         if ∃(Q, W') ∈ R then
11:             R ← R \ {(Q, W')} ∪ {(Q, W' ∪ W)}
12:         else
13:             R ← R ∪ {(Q, W)}
14:         end if
15:     end for
16:     return R
17: end function
```

Case 4: Data produced for activities following the loop In this case, Q_s^* of activities which can be reached from the current loop l have to be checked whether they contain a writer contained in the loop l. In the construction of Q_s, this writer is replaced with the collecting writer w_l, which collects the data written in the loop. If a writer is contained in a loop l_n nested in l, the data of that writer is collected by w_{l_n}. Nevertheless, the data of w_{l_n} has to be collected by w_l. To create the collecting writer w_l, we define $Q_s^{postloop} : \mathcal{A}_{while} \times \mathcal{V} \to \wp(\wp(\mathcal{Q}_\mathcal{V}) \times \wp(\mathcal{A}))$. $Q_s^{postloop}$ returns a set of tuples of queries and writers to all of these queries. The writers returned are writing to variable elements read by an activity after l. Algorithm 18 on the following page presents the implementation. The algorithm iterates over all successors a of the loop (line 3). For each a, the algorithm iterates over the result of Q_s^* (line 4). If there exist writers in the loop l for the activity a, they have to be handled (line 6ff.). Lines 7ff. ensure that collecting writers are used where appropriate. The writers on the current set of queries Q are added to the result in lines 10.

5.2 Treatment of Scopes

Recall from the introduction that the target application for this data flow analysis algorithm restricts data flow into a fault handler of a split scope such that it may only

Algorithm 18 The function $Q_s^{postloop}$, which determines $Q_s(l, v)$ for a reader collecting the data produced by the loop l

```
1:  function Q_s^postloop(l,v)
2:      R ← ∅
3:      for all a ∈ {a | ∃path(l, a) : a ∉ clan(l)} do
4:          for all (Q, W) ∈ Q_s^*(a, v) do
5:              W ← W ∩ clan(l)
6:              if W ≠ ∅ then
7:                  while ∃w ∈ W : l(w) ≠ l do
8:                      W ← W \ {w} ∪ w_∑(w)
9:                  end while
10:                 if ∃(Q, W') ∈ R then
11:                     R ← R \ {(Q, W')} ∪ {(Q, W' ∪ W)}
12:                 else
13:                     R ← R ∪ {(Q, W)}
14:                 end if
15:             end if
16:         end for
17:     end for
18:     return R
19: end function
```

come from the faulting activity. It is propagated at runtime via a coordinator using the coordination protocols in [KL07]. $\mathsf{writes}_\circ$ has been created to reflect that fact. Thus, there is no additional special treatment for fault handlers needed.

For compensation handlers, the data needed in the compensation handler and written outside the compensation handler has to be collected at the end of the belonging scope [Kha07]. Thus, the writers have to be divided into two sets: Writers contained and writers not contained in the compensation handler. This is comparable to the loop cases 1 and 2, where data produced before and in the loop is distinguished. Since the compensation handler itself does not iterate (and is not called multiple times), there is no analogous case for loop case 3, where data produced in a previous iteration needs to be collected. The compensation handler works on local copies of variables, therefore we do not have to deal with data produced by the compensation handler for subsequent activities, as it was the case in loop case 4.

To define the algorithms, we need the function $\mathsf{ch} : \mathcal{A} \to \mathcal{A} \cup \{\bot\}$, which returns the activity a directly nested in the immediate enclosing compensation handler for the given activity:

$$\mathsf{ch} : a \mapsto \begin{cases} a_{ch} & \exists (s, e, a_{ch}) \in \mathsf{HR} : e \in \mathcal{E}_{compensation},\ a \in \mathsf{clan}(a_{ch}) \wedge \\ & \nexists\, b : (s', e', a') \in \mathsf{HR},\ e' \in \mathcal{E}_{compensation},\ a \in \mathsf{clan}(a'), \\ & |\mathsf{clan}(a')| < |\mathsf{clan}(a_{ch})| \\ \bot & \text{otherwise} \end{cases}$$

Data written outside a compensation handler and read inside is collected at the end of the scope the compensation handler belongs to. This data is then stored by the BPEL engine and made available to the compensation handler as soon as it is executed. If the compensation handler is nested in a loop, only data written in the smallest loop containing the handler has to be considered. All other data is already collected by the collecting writers presented in the previous section.

The function $Q_s^{prehandler} : \mathcal{E}_{compensation} \times \mathcal{V} \to \wp(\wp(\mathcal{Q}_\mathcal{V}) \times \wp(\mathcal{A}))$ is returning a set of tuples of queries and writers to all of these queries. The writers returned write to variable elements read in the given compensation handler. Algorithm 19 on the next page presents the implementation. Line 4 merges the Q_s^* of all activities nested in the given compensation handler. Line 7 ensures that only writers not nested in the given compensation handler are treated. If the compensation handler is nested in a loop, the data of writers preceding the loop have already been collected during the handling of the loop. Therefore, all writers outside the enclosing loop are removed (lines 8ff.). The writers introduced by looping are also handled, therefore they get removed in line 11. Data can still be written in loops executed before the end of the scope is reached. Therefore, the collecting writers have to be used where appropriate (lines 12ff).

Algorithm 20 on page 31 presents the function $Q_s^{handler}$ which is used to return Q_s for an activity nested in a compensation handler. $Q_s^{handler}$ ensures that there are no writers

Algorithm 19 The function $Q_s^{prehandler}$, which returns $Q_s(ch, v)$ for a new reader collecting the data at the end of the scope where the compensation handler ch belongs to

1: **function** $Q_s^{prehandler}(ch,v)$
2: Let a_{ch} be the activity executed when the given compensation event occurs.
3: // $(a', ch, a_{ch}) \in \mathsf{HR}$
4: $Q_s^{prehandler\prime} \leftarrow Q_s^{*,\cup}(a_{ch}, v)$
5: $R \leftarrow \emptyset$
6: **for all** $(Q, W') \in Q_s^{prehandler\prime}$ **do**
7: $W' \leftarrow W' \setminus \mathsf{clan}(a_{ch})$
8: **if** $\mathsf{l}(a_{ch}) \neq \bot$ **then**
9: $W' \leftarrow W' \cap \mathsf{clan}(\mathsf{l}(a_{ch}))$
10: **end if**
11: $W \leftarrow \{w \mid w \in W', \mathsf{path}_{\emptyset}(w, a_{ch}) = \mathtt{true}\}$
12: **while** $\exists w \in W : \mathsf{l}(a_{ch}) \neq \mathsf{l}(w)$ **do**
13: $W \leftarrow W \setminus \{w\} \cup \{\mathsf{w}_{\sum}(w)\}$
14: **end while**
15: $R \leftarrow R \cup \{(Q, W)\}$
16: **end for**
17: **return** R
18: **end function**

returned which are not part of the compensation handler (line 4). Lines 5ff. ensure that collecting writers are used where appropriate.

Algorithm 20 The function $Q_s^{handler}$ to determine Q_s for activities nested in compensation handlers

```
 1: function Q_s^handler(a,v)
 2:     R ← ∅
 3:     for all (Q, W) ∈ Q_s^*(a, v) do
 4:         W ← W ∩ clan(ch(a))
 5:         while ∃w ∈ W : l(w) ∈ clan(ch(a)) do
 6:             W ← W \ {w} ∪ w_∑(w)
 7:         end while
 8:         if W ≠ ∅ then
 9:             R ← R ∪ {(Q, W)}
10:         end if
11:     end for
12:     return R
13: end function
```

5.3 The Complete Algorithm

Now, the complete function Q_s can be defined as presented in Algorithm 21. That algorithm uses the function DetermineDirectNesting (Algorithm 22 on the following page) whether a's directly enclosing element, if only loops and compensation handlers are regarded, is a loop or a compensation handler.

Algorithm 21 The function Q_s

```
 1: function Q_s(a,v)
 2:     (nestedInCH, nestedInLoop) ← DetermineDirectNesting(a)
 3:     if nestedInCH then
 4:         return Q_s^handler(a,v)
 5:     else if nestedInLoop then
 6:         return Q_s^↻(a,v)
 7:     else
 8:         return Q_s^*(a, v)
 9:     end if
10: end function
```

Algorithm 22 Determination of the direct enclosing element

```
1: function DetermineDirectNesting(a)
2:     // Check whether a compensation handler or a loop is directly enclosing a
3:     nestedInCH ← false
4:     nestedInLoop ← false
5:     a_ch ← ch(a)
6:     l ← l(a)
7:     if a_ch ∧ l ≠ ⊥ then
8:         if |clan(a_ch)| > |clan(l)| then
9:             nestedInCH ← true
10:        else
11:            nestedInLoop ← true
12:        end if
13:    else if a_ch ≠ ⊥ then
14:        nestedInCH ← true
15:    else if l ≠ ⊥ then
16:        nestedInLoop ← true
17:    end if
18:    return (nestedInCh, nestedInLoop)
19: end function
```

6 Conclusion and Future Work

In this paper, we strongly motivated that algorithms doing data-flow analysis on BPEL processes should be aware of dead path elimination. Our presented algorithm allows for determining def-use edges in BPEL processes, where dead path elimination is activated. We showed how the join conditions can be used to reduce the amount of possible writers for an activity. The presented algorithm is extensible to support arbitrary structured activities, which 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 def-use edges in arbitrary BPEL processes.

The algorithm executed a "Reaching Definitions Analysis" on a BPEL process graph. One aspect in our ongoing work is to modify the lattice to support "Available Expressions Analysis". Another aspect is to investigate whether our treatment of join conditions can be used to handle the reversed flow graph to support the "Very-busy Expressions Analysis" and the "Life-variable Analysis".

Acknowledgments. We thank Steffen Keul, Michele Mancioppi, Daniel Martin, Ralph Mietzner, Simon Moser, Tobias Unger, Jussi Vanhatalo, Matthias Wieland, Andrea Wöhr and Daniel Wutke for the discussions we had on the subject and their valuable feedback. Oliver Kopp is funded by the German Federal Ministry of Education and

Research (project Tools4BPEL, project number 01ISE08).

References

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[BFG05] M. Benedikt, W. Fan, F. Geerts. Xpath satisfiability in the presence of dtds. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 25–36. ACM Press, New York, NY, USA, 2005. doi:10.1145/1065167.1065172.

[BK05] F. van Breugel, M. Koshkina. Dead-path-elimination in bpel4ws. *acsd*, 00:192–201, 2005. doi:10.1109/ACSD.2005.11.

[CC76] P. Cousot, R. Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming, Paris, France*. 1976.

[CDG+03] F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, S. Weerawarana. Business process execution language for web services version 1.1, 2003.

[CKLW03] F. Curbera, R. Khalaf, F. Leymann, S. Weerawarana. Exception handling in the BPEL4WS language. In W. M. P. van der Aalst, A. H. M. ter Hofstede, M. Weske, editors, *BPM 2003*, volume 2678 of *Lecture Notes in Computer Science*, pp. 276–290. Springer, 2003.

[Hei03] T. Heidinger. Statische Analyse von BPEL4WS-Prozemodellen, 2003. Studienarbeit, Humboldt-Universität zu Berlin.

[Kha07] R. Khalaf. Note on syntactic details of split bpel-d business processes. Technical Report Computer Science 2007/02, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Architecture of Application Systems, 2007.

[KKL07] R. Khalaf, O. Kopp, F. Leymann. Maintaining data dependencies across bpel process fragments. In B. J. Krämer, K.-J. Lin, P. Narasimhan, editors, *Service-Oriented Computing - ICSOC 2007*, volume 4749 of *LNCS*, pp. 207–219. Springer, 2007. doi:10.1007/978-3-540-74974-5_17.

[KL06] R. Khalaf, F. Leymann. Role-based decomposition of business processes using BPEL. In *ICWS 2006*, pp. 770–780. IEEE Computer Society, 2006. doi:10.1109/ICWS.2006.56.

[KL07] R. Khalaf, F. Leymann. Coordination protocols for split BPEL loops and scopes. Technical Report 2007/01, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Architecture of Application Systems, 2007.

[LA94] F. Leymann, W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 33(2):326–348, 1994.

[LMP97] J. Lee, S. P. Midkiff, 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*, volume 1366 of *Lecture Notes in Computer Science*, pp. 114–130. Springer, 1997.

[LR00] F. Leymann, D. Roller. *Production Workflow – Concepts and Techniques*. Prentice Hall PTR, 2000.

[MMG+07] S. Moser, A. Martens, K. Görlach, W. Amme, A. Godlinski. 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)*. 2007.

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[NNH04] F. Nielson, H. R. Nielson, C. Hankin. *Principles of Program Analysis*. Springer, 2004.

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0 – OASIS Standard. Technical report, Organization for the Advancement of Structured Information Standards (OASIS), 2007.

[OVA+05] C. Ouyang, H. M. W. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, A. H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL (revised version), 2005.

[W3C99] W3C. XML path language (XPath) version 1.0. W3C Recommendation, 1999.

A Example

In this section, we present an example illustrating the data-flow analysis. Figure 6 presents an example ordering process. The process supports one item per time and charges the amount in four rates. Gold customers get 10% discount, Silver customers get 5% discount and all others any discount. After receiving the order (A) and calculating the appropriate discount (C, D or neither), the order status is updated (E), the order is processed (F), the customer account is billed in 4 weekly installments (while loop W with G,H,I) and a response is sent back acknowledging the order and stating the discount received (H).

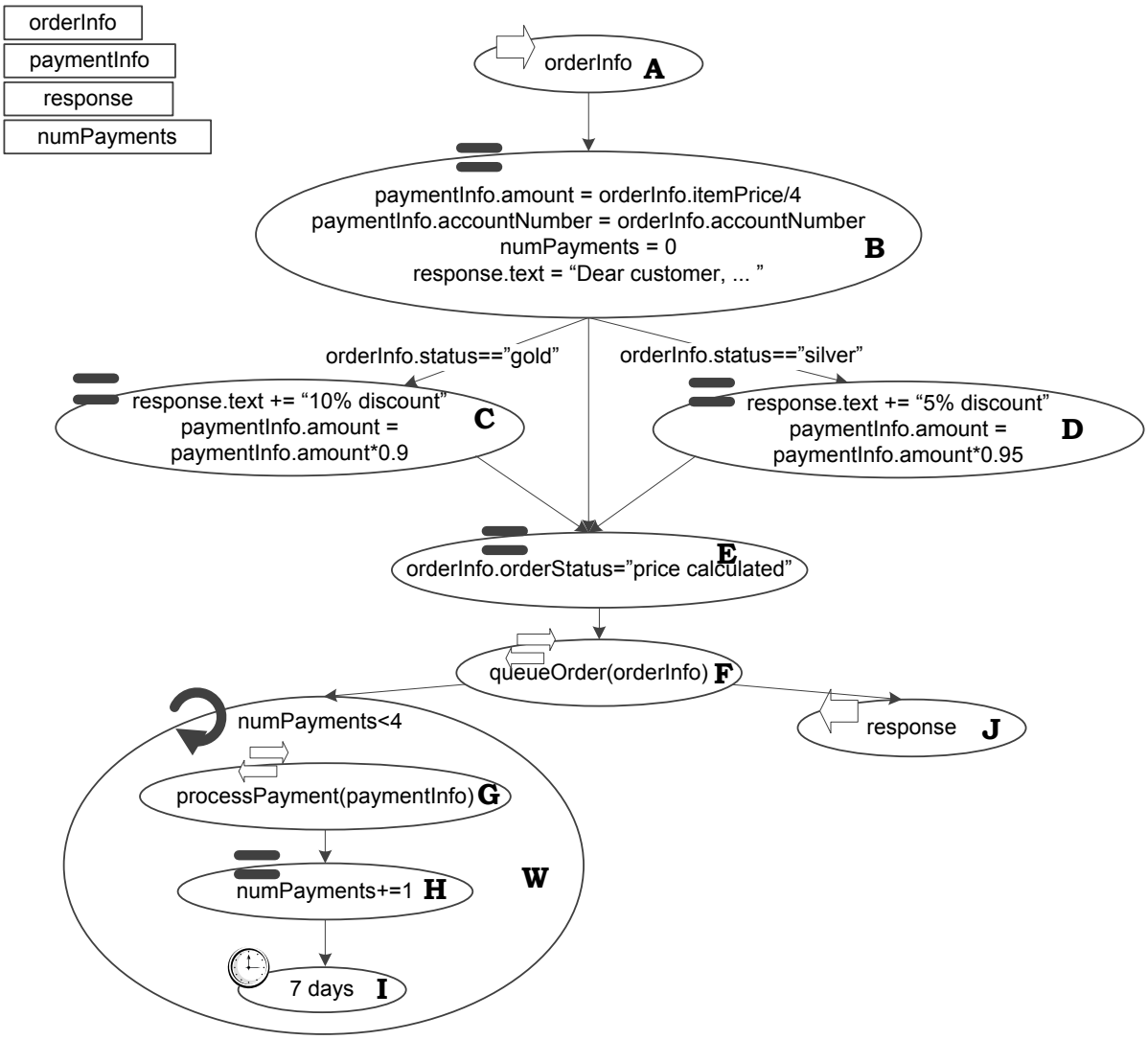

Figure 6: Example process. The upper left corner presents the used variables.

Table 4 presents the reaching definition analysis results. The first two columns contain the activity a and the current variable element v_e. The next four columns contain the value of $\mathsf{poss}(a, v_e)$, $\mathsf{dis}(a, v_e)$, $\mathsf{inv}(a, v_e)$, $\mathsf{mbd}(a, v_e)$. The index on a in the first column denotes whether it is "$\mathsf{poss}_\circ$" or "$\mathsf{poss}_\bullet$".

Table 4: Analysis result

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
$A_\circ$	numPayments	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	orderInfo.accountNumber	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	orderInfo.itemPrice	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	orderInfo.orderStatus	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	oderInfo	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	paymentInfo.accountNumber	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	paymentInfo.amount	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	paymentInfo	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	response.text	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	response	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	numPayments	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	orderInfo.accountNumber	{A}	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	orderInfo.itemPrice	{A}	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	orderInfo.orderStatus	{A}	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	orderInfo	{A}	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	paymentInfo.accountNumber	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	paymentInfo.amount	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	paymentInfo	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	response.text	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	response	$\emptyset$	$\emptyset$	$\emptyset$	`false`

Continued on next page

Table 4: Analysis result

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
$B_\circ$	numPayments	∅	∅	∅	`false`
$B_\circ$	orderInfo.accountNumber	{A}	∅	∅	`false`
$B_\circ$	orderInfo.itemPrice	{A}	∅	∅	`false`
$B_\circ$	orderInfo.orderStatus	{A}	∅	∅	`false`
$B_\circ$	orderInfo	{A}	∅	∅	`false`
$B_\circ$	paymentInfo.accountNumber	∅	∅	∅	`false`
$B_\circ$	paymentInfo.amount	∅	∅	∅	`false`
$B_\circ$	paymentInfo	∅	∅	∅	`false`
$B_\circ$	response.text	∅	∅	∅	`false`
$B_\circ$	response	∅	∅	∅	`false`
$B_\bullet$	numPayments	{B}	∅	∅	`false`
$B_\bullet$	orderInfo.accountNumber	{A}	∅	∅	`false`
$B_\bullet$	orderInfo.itemPrice	{A}	∅	∅	`false`
$B_\bullet$	orderInfo.orderStatus	{A}	∅	∅	`false`
$B_\bullet$	orderInfo	{A}	∅	∅	`false`
$B_\bullet$	paymentInfo.accountNumber	{B}	∅	∅	`false`
$B_\bullet$	paymentInfo.amount	{B}	∅	∅	`false`
$B_\bullet$	paymentInfo	∅	∅	∅	`false`
$B_\bullet$	response.text	{B}	∅	∅	`false`
$B_\bullet$	response	∅	∅	∅	`false`
$C_\circ$	numPayments	{B}	∅	∅	`true`
$C_\circ$	orderInfo.accountNumber	{A}	∅	∅	`true`
$C_\circ$	orderInfo.itemPrice	{A}	∅	∅	`true`
$C_\circ$	orderInfo.orderStatus	{A}	∅	∅	`true`
$C_\circ$	orderInfo	{A}	∅	∅	`true`
$C_\circ$	paymentInfo.accountNumber	{B}	∅	∅	`true`
$C_\circ$	paymentInfo.amount	{B}	∅	∅	`true`
$C_\circ$	paymentInfo	∅	∅	∅	`true`
$C_\circ$	response.text	{B}	∅	∅	`true`
$C_\circ$	response	∅	∅	∅	`true`

Continued on next page

Table 4: Analysis result

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
C•	numPayments	{B}	∅	∅	`true`
C•	orderInfo.accountNumber	{A}	∅	∅	`true`
C•	orderInfo.itemPrice	{A}	∅	∅	`true`
C•	orderInfo.orderStatus	{A}	∅	∅	`true`
C•	orderInfo	{A}	∅	∅	`true`
C•	paymentInfo.accountNumber	{B}	∅	∅	`true`
C•	paymentInfo.amount	{C}	{B}	∅	`false`[4]
C•	paymentInfo	∅	∅	∅	`true`
C•	response.text	{C}	{B}	∅	`false`[4]
C•	response	∅	∅	∅	`true`
D◦	numPayments	{B}	∅	∅	`true`
D◦	orderInfo.accountNumber	{A}	∅	∅	`true`
D◦	orderInfo.itemPrice	{A}	∅	∅	`true`
D◦	orderInfo.orderStatus	{A}	∅	∅	`true`
D◦	orderInfo	{A}	∅	∅	`true`
D◦	paymentInfo.accountNumber	{B}	∅	∅	`true`
D◦	paymentInfo.amount	{B}	∅	∅	`true`
D◦	paymentInfo	∅	∅	∅	`true`
D◦	response.text	{B}	∅	∅	`true`
D◦	response	∅	∅	∅	`true`
D•	numPayments	{B}	∅	∅	`true`
D•	orderInfo.accountNumber	{A}	∅	∅	`true`
D•	orderInfo.itemPrice	{A}	∅	∅	`true`
D•	orderInfo.orderStatus	{A}	∅	∅	`true`
D•	orderInfo	{A}	∅	∅	`true`
D•	paymentInfo.accountNumber	{B}	∅	∅	`true`
D•	paymentInfo.amount	{D}	{B}	∅	`false`
D•	paymentInfo	∅	∅	∅	`true`
D•	response.text	{D}	{B}	∅	`false`
D•	response	∅	∅	∅	`true`

Continued on next page

[4]The existence of a transition condition between C and a succeeding write is reset to `false`.

Table 4: Analysis result

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
$E_\circ$	numPayments	{B}	∅	∅	`false`[5]
$E_\circ$	orderInfo.accountNumber	{A}	∅	∅	`false`
$E_\circ$	orderInfo.itemPrice	{A}	∅	∅	`false`
$E_\circ$	orderInfo.orderStatus	{A}	∅	∅	`false`
$E_\circ$	orderInfo	{A}	∅	∅	`false`
$E_\circ$	paymentInfo.accountNumber	{B}	∅	∅	`false`
$E_\circ$	paymentInfo.amount	{B,C,D}	{B}[6]	∅	`false`
$E_\circ$	paymentInfo	∅	∅	∅	`false`
$E_\circ$	response.text	{B,C,D}	{B}	∅	`false`
$E_\circ$	response	∅	∅	∅	`false`
$E_\bullet$	numPayments	{B}	∅	∅	`false`
$E_\bullet$	orderInfo.accountNumber	{A}	∅	∅	`false`
$E_\bullet$	orderInfo.itemPrice	{A}	∅	∅	`false`
$E_\bullet$	orderInfo.orderStatus	{E}	∅	{A}	`false`
$E_\bullet$	orderInfo	{A}	∅	∅	`false`
$E_\bullet$	paymentInfo.accountNumber	{B}	∅	∅	`false`
$E_\bullet$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
$E_\bullet$	paymentInfo	∅	∅	∅	`false`
$E_\bullet$	response.text	{B,C,D}	{B}	∅	`false`
$E_\bullet$	response	∅	∅	∅	`false`

Continued on next page

[5] $\mathtt{mbd}_\bullet(\text{B}, \text{numPayments})=\mathtt{false}$, $\mathtt{mbd}_\bullet(\text{C}, \text{numPayments})=\mathtt{true}$, $\mathtt{mbd}_\bullet(\text{D}, \text{numPayments})=\mathtt{true}$. Thus $\neg[\![E, \text{numPayments}]\!]^{jc}_{\neg\mathtt{mbd}_\bullet} = \neg(\mathtt{true} \vee \mathtt{false} \vee \mathtt{false}) = \neg(\mathtt{true}) = \mathtt{false}$. Thus $\mathtt{mbd}_\circ(\text{E}, \text{numPayments}) = \mathtt{false}$.

[6] B appears both as possible and disabled writer. It was disabled by C and D, but is still a valid writer if the execution directly reaches E and neither C or E are executed.

Table 4: Analysis result

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
$F_\circ$	numPayments	{B}	∅	∅	`false`
$F_\circ$	orderInfo.accountNumber	{A}	∅	∅	`false`
$F_\circ$	orderInfo.itemPrice	{A}	∅	∅	`false`
$F_\circ$	orderInfo.orderStatus	{E}	∅	{A}	`false`
$F_\circ$	orderInfo	{A}	∅	∅	`false`
$F_\circ$	paymentInfo.accountNumber	{B}	∅	∅	`false`
$F_\circ$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
$F_\circ$	paymentInfo	∅	∅	∅	`false`
$F_\circ$	response.text	{B,C,D}	{B}	∅	`false`
$F_\circ$	response	∅	∅	∅	`false`
$F_\bullet$	numPayments	{B}	∅	∅	`false`
$F_\bullet$	orderInfo.accountNumber	{A}	∅	∅	`false`
$F_\bullet$	orderInfo.itemPrice	{A}	∅	∅	`false`
$F_\bullet$	orderInfo.orderStatus	{E}	∅	{A}	`false`
$F_\bullet$	orderInfo	{A}	∅	∅	`false`
$F_\bullet$	paymentInfo.accountNumber	{B}	∅	∅	`false`
$F_\bullet$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
$F_\bullet$	paymentInfo	∅	∅	∅	`false`
$F_\bullet$	response.text	{B,C,D}	{B}	∅	`false`
$F_\bullet$	response	∅	∅	∅	`false`
$G_\circ$	numPayments	{B,H}	∅	{B}	`false`
$G_\circ$	orderInfo.accountNumber	{A}	∅	∅	`false`
$G_\circ$	orderInfo.itemPrice	{A}	∅	∅	`false`
$G_\circ$	orderInfo.orderStatus	{E}	∅	{A}	`false`
$G_\circ$	orderInfo	{A}	∅	∅	`false`
$G_\circ$	paymentInfo.accountNumber	{B}	∅	∅	`false`
$G_\circ$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
$G_\circ$	paymentInfo	∅	∅	∅	`false`
$G_\circ$	response.text	{B,C,D}	{B}	∅	`false`
$G_\circ$	response	∅	∅	∅	`false`

Continued on next page

Table 4: Analysis result

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
G$_\bullet$	numPayments	{B,H}	∅	{B}	`false`
G$_\bullet$	orderInfo.accountNumber	{A}	∅	∅	`false`
G$_\bullet$	orderInfo.itemPrice	{A}	∅	∅	`false`
G$_\bullet$	orderInfo.orderStatus	{E}	∅	{A}	`false`
G$_\bullet$	orderInfo	{A}	∅	∅	`false`
G$_\bullet$	paymentInfo.accountNumber	{B}	∅	∅	`false`
G$_\bullet$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
G$_\bullet$	paymentInfo	∅	∅	∅	`false`
G$_\bullet$	response.text	{B,C,D}	{B}	∅	`false`
G$_\bullet$	response	∅	∅	∅	`false`
H$_\circ$	numPayments	{B,H}	∅	{B}	`false`
H$_\circ$	orderInfo.accountNumber	{A}	∅	∅	`false`
H$_\circ$	orderInfo.itemPrice	{A}	∅	∅	`false`
H$_\circ$	orderInfo.orderStatus	{E}	∅	{A}	`false`
H$_\circ$	orderInfo	{A}	∅	∅	`false`
H$_\circ$	paymentInfo.accountNumber	{B}	∅	∅	`false`
H$_\circ$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
H$_\circ$	paymentInfo	∅	∅	∅	`false`
H$_\circ$	response.text	{B,C,D}	{B}	∅	`false`
H$_\circ$	response	∅	∅	∅	`false`
H$_\bullet$	numPayments	{H}	∅	{B}	`false`
H$_\bullet$	orderInfo.accountNumber	{A}	∅	∅	`false`
H$_\bullet$	orderInfo.itemPrice	{A}	∅	∅	`false`
H$_\bullet$	orderInfo.orderStatus	{E}	∅	{A}	`false`
H$_\bullet$	orderInfo	{A}	∅	∅	`false`
H$_\bullet$	paymentInfo.accountNumber	{B}	∅	∅	`false`
H$_\bullet$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
H$_\bullet$	paymentInfo	∅	∅	∅	`false`
H$_\bullet$	response.text	{B,C,D}	{B}	∅	`false`
H$_\bullet$	response	∅	∅	∅	`false`

Continued on next page

Table 4: Analysis result

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
$I_\circ$	numPayments	{H}	∅	{B}	`false`
$I_\circ$	orderInfo.accountNumber	{A}	∅	∅	`false`
$I_\circ$	orderInfo.itemPrice	{A}	∅	∅	`false`
$I_\circ$	orderInfo.orderStatus	{E}	∅	{A}	`false`
$I_\circ$	orderInfo	{A}	∅	∅	`false`
$I_\circ$	paymentInfo.accountNumber	{B}	∅	∅	`false`
$I_\circ$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
$I_\circ$	paymentInfo	∅	∅	∅	`false`
$I_\circ$	response.text	{B,C,D}	{B}	∅	`false`
$I_\circ$	response	∅	∅	∅	`false`
$I_\bullet$	numPayments	{H}	∅	{B}	`false`
$I_\bullet$	orderInfo.accountNumber	{A}	∅	∅	`false`
$I_\bullet$	orderInfo.itemPrice	{A}	∅	∅	`false`
$I_\bullet$	orderInfo.orderStatus	{E}	∅	{A}	`false`
$I_\bullet$	orderInfo	{A}	∅	∅	`false`
$I_\bullet$	paymentInfo.accountNumber	{B}	∅	∅	`false`
$I_\bullet$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
$I_\bullet$	paymentInfo	∅	∅	∅	`false`
$I_\bullet$	response.text	{B,C,D}	{B}	∅	`false`
$I_\bullet$	response	∅	∅	∅	`false`
$J_\circ$	numPayments	{B}	∅	∅	`false`
$J_\circ$	orderInfo.accountNumber	{A}	∅	∅	`false`
$J_\circ$	orderInfo.itemPrice	{A}	∅	∅	`false`
$J_\circ$	orderInfo.orderStatus	{E}	∅	{A}	`false`
$J_\circ$	orderInfo	{A}	∅	∅	`false`
$J_\circ$	paymentInfo.accountNumber	{B}	∅	∅	`false`
$J_\circ$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
$J_\circ$	paymentInfo	∅	∅	∅	`false`
$J_\circ$	response.text	{B,C,D}	{B}	∅	`false`
$J_\circ$	response	∅	∅	∅	`false`

Continued on next page

Table 4: Analysis result

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
J$_\bullet$	numPayments	{B}	∅	∅	`false`
J$_\bullet$	orderInfo.accountNumber	{A}	∅	∅	`false`
J$_\bullet$	orderInfo.itemPrice	{A}	∅	∅	`false`
J$_\bullet$	orderInfo.orderStatus	{E}	∅	{A}	`false`
J$_\bullet$	orderInfo	{A}	∅	∅	`false`
J$_\bullet$	paymentInfo.accountNumber	{B}	∅	∅	`false`
J$_\bullet$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
J$_\bullet$	paymentInfo	∅	∅	∅	`false`
J$_\bullet$	response.text	{B,C,D}	{B}	∅	`false`
J$_\bullet$	response	∅	∅	∅	`false`
W$_\circ$[7]	numPayments	{B,H}[8]	∅	{B}	`false`
W$_\circ$	orderInfo.accountNumber	{A}	∅	∅	`false`
W$_\circ$	orderInfo.itemPrice	{A}	∅	∅	`false`
W$_\circ$	orderInfo.orderStatus	{E}	∅	{A}	`false`
W$_\circ$	orderInfo	{A}	∅	∅	`false`
W$_\circ$	paymentInfo.accountNumber	{B}	∅	∅	`false`
W$_\circ$	paymentInfo.amount	{B,C,D}	{B}	∅	`false`
W$_\circ$	paymentInfo	∅	∅	∅	`false`
W$_\circ$	response.text	{B,C,D}	{B}	∅	`false`
W$_\circ$	response	∅	∅	∅	`false`

[7]The results for W$_\circ$ and $W_\bullet$ are equal. The flow activity directly nested in W is not explicitly listed. The values for that activity are also equal.

[8]B appears both as possible and disabled writer. This is valid: If the while loop is never executed, B is a possible writer. If the while loop is executed, it gets disabled by H