Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 2726

Extended Data-flow Analysis on BPEL Processes

Sebastian Breier

Course of Study:	Software Engineering
Examiner:	Prof. Dr. F. Leymann
Supervisor:	Dipl.-Inf. Oliver Kopp
Commenced:	January 14, 2008
Completed:	July 22, 2008
CR-Classification:	F.3.2, H.4.1

Contents

1 Introduction **7**
1.1 Possible Application: Split BPEL Processes 7

2 Web Services Business Process Execution Language 2.0 **11**
2.1 Basic Activities 12
2.2 Container Activities 14
2.3 Graphical Conventions 18

3 Data Flow Analysis **21**
3.1 Concurrent Single Static Assignment Form 22
3.2 Petri Nets 23
3.3 Future Work 25

4 Algorithm Fundamentals **27**
4.1 Variables 28
4.2 Writer State 28
4.3 Activity and Link Flags 29
4.4 Events 31
4.5 Challenges for Data Flow by DPE 31
4.6 Repeatable Activities 34
4.7 Restrictions 34

5 Motivating Example **37**

6 Equivalence Classes **41**
6.1 Basic Activities 42
6.2 Compensation Activities 43
6.3 Container Activities 43
6.4 Simple Container Activities 44
6.5 `scope` Activity 45
6.6 Complex Container Activities 45
6.7 Non-Repeatable Container Activities 45
6.8 Repeatable Container Activities 46

7 The Algorithm **47**
7.1 General Activity Handling . . . 48
7.2 Data Handling . . . 50
7.3 Begin and End of Analysis . . . 63
7.3.1 Analysis Structure . . . 63
7.4 Basic Activities . . . 64
7.5 `flow` Activity . . . 65
7.6 BPEL Links . . . 67
7.7 `sequence` Activity . . . 69
7.8 Loop Activities . . . 70
7.9 `pick` Activity . . . 71
7.10 `if` Activity . . . 73
7.11 `scope` Activity . . . 75
7.11.1 Fault Handlers . . . 77
7.11.2 Event Handlers . . . 80
7.11.3 Termination Handler . . . 81
7.11.4 End of `scope` Handling . . . 84
7.11.5 Compensation Handler . . . 85
7.12 Revisiting the initial example . . . 88
7.13 Conclusion . . . 93

8 Implementation **95**

9 Summary and Outlook **97**

A Notations **99**

Bibliography **103**

List of Figures

1.1 Decomposition of BPEL processes 8

2.1 Dead Path Elimination 16
2.2 Graphical conventions 18

3.1 Example variable usage from BPEL2oWFN 24

4.1 Example for failing visited flag 30
4.2 New started and visited flags 31
4.3 DPE issue concerning scoping 33

5.1 Motivating example 38

7.1 Data flow scoping 54
7.2 Re-enabling writers in a sequence 55
7.3 Function overWrite 57
7.4 mbd propagation through a container activity that has two data flow paths 58
7.5 Emergence of duplications (OR) with inv 60
7.6 Emergence of duplications (OR) with dis 61
7.7 Emergence of duplications (XOR) 62
7.8 Analysis structure 64
7.9 Variations of cross-boundary links 68
7.10 Accumulation of data flow edges in a basic scope 78
7.11 Duplicates in a Fault Handler 79
7.12 scope overview 83
7.13 Compensation activities 87
7.14 Motivating example 88

List of Tables

7.1 Analysis result 93

A.1 Notations used 101

List of Listings

7.1 Outline of a `pick` activity 71
7.2 Outline of an `if` activity 74
7.3 Outline of a `scope` activity 76

List of Algorithms

1 Handling of an activity 49
2 Handling of activity successors 51
3 Data merging 52
4 Data overwriting 56
5 Duplication purging (multiple paths) 59
6 Duplication purging (single path) 61
7 Analysis of a given BPEL process 63
8 Handling a basic activity 64
9 Handling a `flow` activity 65
10 Handling the end of a `flow` activity 66
11 Handling a link 67
12 Data overwriting without ending dead paths 69
13 Handling the end of a sequence activity 70
14 Handling looping or ending `while`/`repeatUntil`/`forEach` loops 71
15 Handling `pick` activities 72
16 Handling ending pick activities 73
17 Handling `if` activities 74
18 Handling ending `if` activities 75
19 Analysis of `scope` handlers 76
20 Analysis of fault handlers 77
21 Analysis of event handlers 80
22 Analysis of termination handler 81
23 Handling ending analysis of a `scope` activity 84
24 Analysis of a Compensation handler 86

Chapter 1

Introduction

The increase of business transactions over the Internet (money transfer, travel booking, stock exchange, etc.) also increases software complexity between and inside companies. To simplify the coupling of software systems, the idea of the Service-Oriented Architecture (SOA) has emerged. A Service-Oriented Architecture consists of distributed, heterogeneous pieces of software that offer certain services. These services can be anything feasible in software.

One realization of an SOA is the "Web services platform". The technologies are mainly based on XML, such as the Web Service Description Language (WSDL). The standards facilitate the description, implementation and message exchange and enforce quality of service. The service descriptions of WSDL describe services offered and the communications protocols used. More information on Web Services can be found in [WCL+05].

The Business Process Execution Language ("WS-BPEL" or "BPEL") is an XML-based programming language used to orchestrate behavior between Web services to form a business process. It has been developed by IBM, Microsoft, BEA, SAP, Siebel and others. It has been standardized by OASIS[1] in 2007. The WS-BPEL 2.0 standard can be found in [OAS07]. An overview of BPEL and its use can be found in [KKL06].

1.1 Possible Application: Split BPEL Processes

With the growth of business processes and the need for companies to stay competitive, outsourcing has been found as a viable method to continue to achieve business goals. Outsourcing of business processes can involve partitioning an existing business process. If the business process is implemented in BPEL, this requires the BPEL process to be partitioned and the parts distributed to the business partners. A problem that has been intensely researched in [KL06], [KKL07a] and [Kha08] is the agile maintenance of operational semantics of partitioned BPEL processes.

A common feature for partitioning algorithms is the requirement of data flow analysis. Shared variables in BPEL necessitate the creation of explicit data links between process partitions, if the partitioning should be efficient. Data flow algorithms for BPEL have already been proposed and

[1] Organization for the Advancement of Structured Information Standards, `http://www.oasis-open.org/`

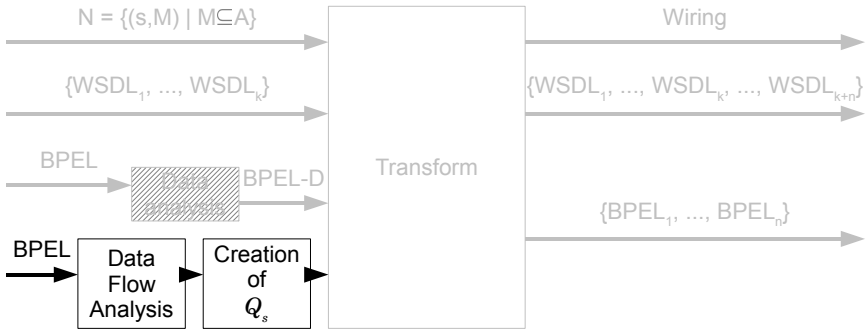


Figure 1.1: Decomposition of BPEL processes

implemented, see Chapter 3. This work presents an algorithm that follows the operation of a BPEL navigator on a BPEL process more closely than other algorithms.

This work can be applied to the decomposition of BPEL processes into separate partitions.

There have been two approaches to this challenge:

1. Convert BPEL to BPEL-D, then partition the BPEL-D process ([KL06]). BPEL-D is a variation of BPEL with explicit instead of implicit data links.
2. Use the results of data flow analysis to create Q_s data necessary for the partitioning process ([KKL07a])

The transformation of BPEL to BPEL-D has been discussed and researched thoroughly in [Kha08]. The application of data flow analysis results to the partitioning process via Q_s data has been done in the technical report on which this thesis is based, namely [KKL07c].

The various approaches can be seen in Figure 1.1. The original diagram is from [KL06], which presents the BPEL-D approach. The updated diagram highlights the updated approach. The presented data flow analysis can be used on the original BPEL process. The results of the data flow analysis have to be converted into the Q_s data, a process which is in part available in [KKL07c]. The Q_s data can be used in [KKL07a] to partition the BPEL process.

The work at hand is structured into the following chapters:

- **Chapter 2 "Web Services Business Process Execution Language 2.0"** provides an introduction and overview over BPEL
- **Chapter 3 "Data Flow Analysis"** describes data flow analysis in general, and more detailed in the application of the Concurrent Single Static Assignment Form and petri nets
- **Chapter 4 "Algorithm Fundamentals"** provides an introduction to the fundamental concepts of the presented data flow algorithm
- **Chapter 5 "Motivating Example"** presents a motivating example that will be used as a starting point of explaining the analysis algorithm

- **Chapter 6 "Equivalence Classes"** groups BPEL activities into a hierarchy of classes of equivalently handled constructs, which explains more of the creation of the algorithm
- **Chapter 7 "The Algorithm"** presents the actual data flow algorithm and explains all the details necessary to grasp the analysis
- **Chapter 8 "Implementation"** provides an overview over the implementation of the algorithm
- **Chapter 9 "Summary and Outlook"** concludes the work

Chapter 2

Web Services Business Process Execution Language 2.0

The Web Services Business Process Execution Language 2.0 (BPEL for short) is a "workflow-based language that aggregates services by choreographing service interaction" [WCL+05] in XML. According to [Wes07], "[a] business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations." This connotes that BPEL is a programming language used to describe activities to produce goods, or in this case services, for a customer.

BPEL has two ways to describe business processes: Executable and abstract business processes. An abstract business process leaves out details of the process, focusing on the broader interaction. "Interaction" in this case is the communication between a BPEL process and a Web service. Executable business processes are actual code, defining interactions detailed enough for execution by a BPEL engine. This work is about data flow in executable business processes and does not discuss any details of abstract business processes.

There are technologies in place for interaction in business processes already, namely:

- **HTTP:** A widely-used communications protocol for networks
- **SOAP:** An XML encoding for sending and receiving messages through a network
- **WSDL:** Web Services Description Language, an XML format used for describing interfaces of Web services
- **UDDI:** Universal Description, Discovery and Integration, an infrastructure to discover offered services
- **XML Schema:** A language describing valid schemas of XML documents, used for message exchanges

However, more logic is needed to combine services. BPEL's executable business processes are one way to provide this logic.

BPEL applies additional technologies:

- **XPath and XSLT:** XML Path Language and Extensible Stylesheet Language Transformations, both applied to XML messages and variables in BPEL for purposes such as reading, writing and transformation.

BPEL processes are run on a BPEL engine. A BPEL engine is a software system providing the platform and framework for BPEL processes to use.

The commands of the BPEL language are called "activities". Activities are partitioned into two groups: Basic activities and container activities.

Basic activities are activities which contain no other activities.

Container activities are called "structured activities" in the BPEL standard. I use the term "container activities" because it is helpful to think of these activities as "containers" when discussing data flow.

By combining block structures and graph structures as container activities, BPEL becomes a powerful language with possibilities beyond traditional programming languages.

The following sections cover these topics:

- **Section 2.1 "Basic Activities"** and **Section 2.2 "Container Activities"** list the BPEL activities
- **Section 2.3 "Graphical Conventions"** explains the graphical conventions used in this work

2.1 Basic Activities

`invoke`

`invoke` a service operation, specified by `partnerLink`, `portType` and `operation`. A `partnerLink` encapsulates the structure of a the relationship between BPEL and a Web service through a Web service interface. A `portType` represents a set of operations available from the `partnerLink` and other endpoints. An `operation` is an action supported by the Web service. More information on Web service terminology can be found in [OAS07] and [Wor01].

Variables will be read from the `inputVariable` and `fromVariable` and written to the `outputVariable` and `toVariable`.

The operation invocation can be a one-way invocation or a request-response invocation. If the operation is one-way and no response is expected, the `inputVariable` or `fromVariable` is needed only. For request-response operations, `invoke` waits for a reply and an additional `outputVariable` or `toVariable` is used.

The `invoke` activity can have fault and compensation handlers that would normally be specified in a `scope`. This is equivalent to a `scope` directly containing the `invoke` activity, having the specified handlers.

receive

Receive a message from the given partnerLink, portType and operation. Writes to the specified variable or toVariable.

reply

Reply to a message that has been received earlier. The reply is taken from the specified variable or fromVariable.

reply is the activity corresponding to the end of a request-response operation started by a remote invoke activity.

assign

assign copies data between variables and other constructs. Variables can be validated (attribute validate). Data can be copied from a source to a target, each being one of the following:

- A variable using a query
- A variable property
- A partnerLink
- Any expression in a specified expressionLanguage
- A literal expression

throw

Generate a fault that will be caught by the fault handlers, or, if no fault handlers are defined, to the BPEL engine. The faultName specifies the type of fault to be thrown. A faultVariable containing additional data can be specified.

wait

Wait for a specified duration or until a certain date/time. Typical usage is the start of an activity at a specified point in time.

empty

No operation. Can be used for synchronizing activities through a control flow join, or to suppress faults (by using empty in a fault handler).

extensionActivity

This activity is used for extensions to the BPEL language that can be made available by the BPEL engine. Activities supplied by BPEL extensions have to be placed inside this activity. This work does not analyze the `extensionActivity`.

exit

`exit` ends the running BPEL process. All activities will be immediately stopped. No termination, fault or compensation handlers are run.

rethrow

`rethrow` can be used inside a fault handler only. It passes a fault down the scoping hierarchy to the next fault handler or the BPEL engine. Changes to the fault data have to be ignored, the fault must be thrown the same as it was caught.

compensate

The `compensate` activity runs the compensation handler of all contained `scopes`, the order of compensation being the reverse finishing order. `compensate` can be called from fault, compensation and fault handlers only.

compensateScope

This activity runs the compensation handler of the specified `scope` (`target`). `compensateScope` can be called from fault, compensation and fault handlers only.

validate

This activity checks a specified BPEL `variable` against its XML definition.

2.2 Container Activities

sequence

Container activity that executes the contained activities in the order they appear in the BPEL process.

`if`

Container activity that runs exactly one of the given activities, depending on the result of a specified condition.

`while`

Container activity that executes the given contained activity repeatedly while a specified condition is `true`.

`repeatUntil`

Container activity that runs the contained activity until a specified condition gets `true`.

`pick`

Container activity. This waits for a message to arrive on the specified `operation`. A timeout can be defined for the messages to arrive in, after which an alternative operation will be performed.

There has to be at least one `onMessage` element.

`flow`

In a `flow` container activity, contained activities are run concurrently. They can be synchronized and ordered through BPEL links.

BPEL links can not form control cycles. That is, the target of a link may not be a preceding activity of the link source in any way.

BPEL links have a `transitionCondition` in a specified `expressionLanguage` that will yield a `true` or `false` result. If no `transitionCondition` is specified, it is `true` by default. The `transitionCondition` can refer to any data available to the link, such as BPEL variables. The link status is the status of the evaluation of the transition condition. This can be `true`, `false` or `unset`.

Each BPEL activity has a `joinCondition` in a specified `expressionLanguage`. The statuses of incoming BPEL links at an activity are evaluated through the `joinCondition`. The `joinCondition` can access the link status only. This will yield a `true` or `false` result. If no `joinCondition` is specified, a logical OR of all incoming links is the default.

If the `process` has its `suppressJoinFailure` attribute set to “no”, a `false joinCondition` throws a `bpel:joinFailure` fault. This is the default. This case can be handled like a regular `scope` fault using a fault handler. Analyzing faults and fault handlers is covered in Section 7.11 on page 75 of this work.

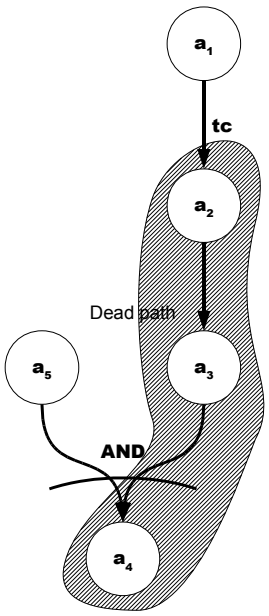


Figure 2.1: Dead Path Elimination

When the `suppressJoinFailure` attribute is set to "yes", no `bpel:joinFailure` fault is thrown. Instead, the `joinCondition` expression determines the execution of an activity. An activity with a `joinCondition` evaluated to `false` will not execute. The condition of execution of contained activities—if the activity is a container activity—is set to `false`, even if their `joinCondition` evaluates to `true`. All outgoing links of activities with a `false` condition of execution will be set to `false`, even if their `transitionCondition` evaluates to `true`. The `joinCondition` of an activity will not be determined until all incoming links have a status of either `true` of `false`, but not while at least one is `unset`. This has the effect of transitively traversing the path of BPEL links starting at a `false` `joinCondition`, disabling all the activities on that path, until a `true` `joinCondition` is reached. This effect is called "Dead Path Elimination" (DPE).

Dead Path Elimination is necessary to render evaluation of the `joinCondition` of activities possible. Consider a dead path, that is, a series of activities chained by BPEL links, with one activity having a `false` `joinCondition` (Figure 2.1). The last activity in the path has a second incoming link that is not in a dead path. Without DPE, evaluation of the `joinCondition` of the last activity might wait forever for the first link status to become either `true` or `false`. With DPE, the dead path is eliminated by setting all `transitionConditions` to `false` successively.

Dead Path Elimination imposes a bigger challenge on data flow analysis. For a detailed account on these challenges, see Section 4.5 on page 31.

Note that the dead path propagates through BPEL links and the containment hierarchy only. An activity contained in a `sequence`, having a disabled predecessor, still executes, as long as the `sequence` itself executes.

BPEL links can not cross the boundary of repeatable constructs—`while`, `repeatUntil`, `forEach`, `eventHandlers`—or of a `compensationHandler`. They can be used inside these constructs, but must have both source and target inside it.

BPEL links can not cross the boundary of a fault handler (`catch` and `catchAll`) or of a termination handler, if the target activity lies inside the handler. For these constructs, the links must be outbound only, or contained completely inside.

`forEach`

Container activity that executes the contained `scope` the amount of times specified by `startCounterValue` and `finalCounterValue`. Execution can be parallel or sequential, specified by the `parallel` attribute. A `completionCondition` can be specified that ends execution of still running branches when a certain condition has been fulfilled.

In the case of ending execution of the `forEach` still running branches are terminated and their termination handlers called.

`scope`

A `scope` is a container activity that enables data scoping in a BPEL process. It can contain its own set of `partnerLinks`, `messageExchanges`, `variables`, `correlationSets`, fault, compensation, termination and event handlers. These elements are valid for directly and deeply contained activities of the `scope` only.

Fault handlers are executed for the appropriate fault when a fault is thrown inside the `scope`. In that case, the `scope` is set to have faulted and no compensation handler is installed. Faults thrown inside the fault handler are propagated to the next enclosing `scope`, or the BPEL `process`. If no fault handler is specified, the default fault handler will sequentially execute compensation for contained `scopes` and then `rethrow` the fault to the containing `scope` or `process`.

The compensation handler is used to undo (*compensate*) the work done in the `scope`. A compensation handler for a `scope` is installed upon successful ending of the `scope`. If a `scope` has faulted or is terminated, no compensation handler is installed. The compensation handler has access to the variable states at the time the `scope` finished execution. When the compensation handler is called to compensate from the `compensate` or `compensateScope` activity, all writes to variables are "live". If no compensation handler is specified, the default compensation handler is installed, which will start compensation of all contained `scopes` in reverse finish order. Running `compensate` or `compensateScope` on an already compensated `scope` has no effect, i. e. it is equivalent to running an `empty` activity. If a `scope` is inside a repeatable construct—a `while`, `repeatUntil` or sequential `forEach`—each `scope` instance has its own compensation handler instance.

The termination handler is called whenever the `scope` is forced to terminate. This does not include a fault in the scope. The termination handler executes after all running activities have been terminated. Some short-lived activities are allowed to complete without being interrupted, such as `assign` activities (details on [OAS07, p. 135 f.]).

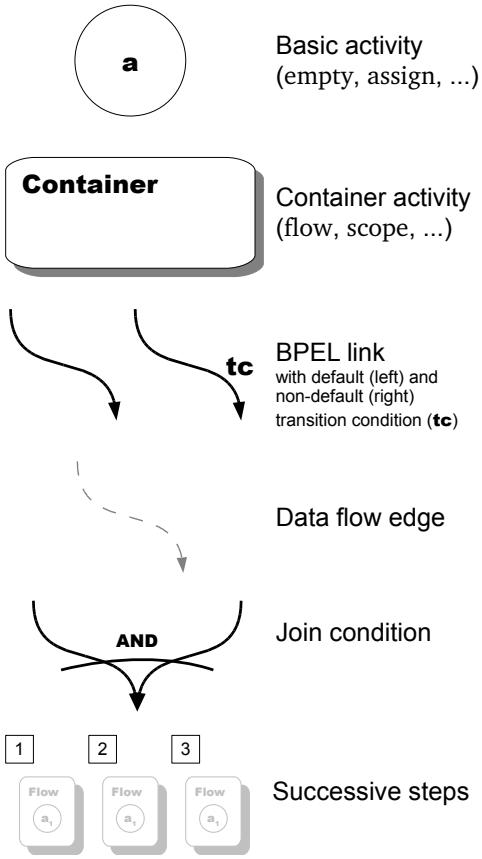


Figure 2.2: Graphical conventions for diagrams

Event handlers are used to react to events (`onMessage` and `onAlarm`) happening while a `scope` is executing. They are enabled when the associated `scope` starts execution. They are disabled when execution of the main contained activity stops.

If a `scope` has its `isolated` attribute set to "yes" (default "no"), it is guaranteed that the results of parallel execution of two isolated `scopes` is the same as of a serialized execution.

2.3 Graphical Conventions

Figure 2.2 depicts the graphical conventions for diagrams used in this work.

Basic activities are displayed as a circle containing the type of the activity. The type can be abbreviated to save space, like "ε" for an `empty` activity, or "w" for a `write` activity. In cases where more than one activity of the same type is displayed, activities are numbered, like "ε_1" for the first `empty` activity, or "w_2" for the second `write` activity.

Container activities are portrayed as rounded, shadowed rectangles. The type of container is specified in the upper left corner. As with basic activities, these can be abbreviated and numbered.

BPEL links are portrayed as strong, black arrows. A default (`true`) join condition of a BPEL link will never be displayed. If the transition condition of a BPEL link is different than the default, it is specified by the letters “tc” next to the link.

Data flow edges are portrayed as light, dashed, gray arrows.

Join conditions are solid, strong curves over the incoming BPEL links. The default join condition is a Boolean *OR* over all incoming links and is left out of diagrams. In cases where a distinction between the default and another join condition is emphasized, the default join condition is shown.

When a figure shows multiple similar images to depict successive steps of an algorithm, the steps are set as solid boxes in the upper left of the images.

Chapter 3

Data Flow Analysis

Data Flow Analysis is an important part of compiler construction. Source code compilers are able to optimize software by implementing program analysis. Program analysis is the process of automatically determining features and attributes of a given software. More information on general program analysis can be found in [ALSU06] and [NNH99].

Program analysis is usually applied to a graph called the "program flow graph". This graph is generated by a parser operating on the source code. The nodes of the graph represent all the commands or a part of a program only. Special nodes, such as entry and exit of a function, or procedure calls, can be added. The edges of the graph can represent any relationship between nodes, like control dependencies. In the case of data flow analysis, at least one type of edge represents data flow relationships.

Data flow is the relationship of data writing and reading in a programming language. Data flow analysis, a part of program analysis, is the analysis of software with the intent of determining data flow. In many cases, the most interesting relationship is the "Def-Use" relationship.

The "Def-Use" relationship consists of two commands in the analyzed software, with the second command being executed chronologically after the first command. The first command writes to the variable. The second command reads from that variable. The actual manifestations of such a relationship are "data flow edges". They are edges between the nodes in the program flow graph.

The following sections cover these topics:

- **Section 3.1 "Concurrent Single Static Assignment Form"** offers an overview over the Single Static Assignment Form and its concurrency extension
- **Section 3.2 "Petri Nets"** discusses petri nets in data flow analysis
- **Section 3.3 "Future Work"** discusses petri nets in data flow analysis

There are possibilities of analyzing the BPEL control flow and data flow that do not depend on the *traditional* approaches of the Concurrent Single Static Assignment Form or petri nets. Those possibilities usually involve parsing and analyzing the BPEL process in a highly BPEL-dependent manner, whereas the traditional approaches are mainly language-independent.

[Hei03] creates a model for static analysis of BPEL processes that is similar to the presented algorithm, but less accurate. It does not analyze control flow at all and it is based on BPEL 1.1. However, the results are small networks that offer a diverse set of analyses.

[Sch07] offers a model and implementation for static analysis of XPath queries in BPEL. The model can be used to determine paths in control flow dependent on XPath queries.

The presented algorithm in this thesis is, like the two references mentioned last, an algorithm for static analysis that offers a BPEL-dependent, but easy to grasp way to analyze data flow.

3.1 Concurrent Single Static Assignment Form

To optimize the representation of data flow, data flow analysis can be implemented using the "Single Static Assignment Form" (see [CFR+91]). Algorithms not using this form might create a high amount of data flow edges, especially when variable usage is in different parts of a branching construct. At the point of variable usage, users of the data flow analysis result will then need to handle a huge amount of possible variable writers. To eliminate many of these edges, additional nodes are added to the program flow graph. These nodes represent the branching and re-joining of program control flow. Additionally, every variable setting is indexed incrementally. The variable usage is changed to reflect the index. At a node with re-joining control flow, a new indexed variable is created that gets set to the result of a special function. This function is "ϕ". Its arguments are all the indexed variables from the joined branches, like "$v_2 \leftarrow \phi(v_0, v_1)$". With this algorithm, each indexed variable has one writer only, thereby decreasing complexity of the data flow graph. Multiple possible writers are collapsed into the ϕ function.

Through the use of the Single Static Assignment Form, many code optimizations and queries on the code are possible. A simple example is the check for uninitialized variable access. A special node with the assignment "$v_0 \leftarrow \mathsf{uninit}$" can be created. If, through the data flow using ϕ functions, a variable reader is able to read from v_0, uninitialized variable access is in principle possible. However, due to SSA being a static analysis, the access might never actually be possible at runtime, even if it was (statically) predicted.

For programs containing concurrency, an extension of SSA has been proposed by [LMP97]. In branching algorithms, it is not possible to statically determine the outcome of a choice correctly. Additionally, in algorithms featuring concurrency, it is not possible to determine the outcome of parallel writes to the same variable. In Concurrent Single Static Assignment Form (CSSA), code blocks are marked as being concurrent. At a synchronization point between code blocks, a node containing a "π" function is added. The π function removes the requirement for every variable access to link to each concurrent block in the data flow graph, similar to the ϕ function.

The Concurrent Single Static Assignment Form has already been applied to both BPEL ([God06]) and XPath ([Gör08]). A shorter discussion can be found in [MMG+07].

There are several reasons for not choosing CSSA for the current work. The most obvious is that it has already been extensively covered for BPEL. Another reason is that there is additional complexity in creating the graph for CSSA and also for evaluating it. The algorithms creating the graph are difficult

to grasp. The result is a graph still containing ϕ and π nodes, which is both different from the original program flow graph, as well as not immediately useful. The result still needs to be processed or at least mapped back to the BPEL process to increase usefulness. The work at hand, as will be shown, follows the program flow graph closely, like a BPEL navigator. The result is an immediately usable graph of BPEL activities connected by data flow edges.

3.2 Petri Nets

A *petri net* is a data structure of places, transitions, edges and markers. A place is a state of a part of the system, such as "started" or "running". A marker on a place declares that the marked place is active and one part of the current system state. Markers are also called "tokens" in some references. A transition is an action that can change the system from one state to another, by transferring markers between connected places. Edges are directed connections between places and transitions.

Many rules can be added to a petri net to convey semantics of a system. Places can contain any number of markers, and edges can "cost" any number of markers to traverse them. A transition can only transfer the markers (= *fire*), if the places connected by incoming edges contain enough markers to cover the cost, and if the places connected by outgoing edges can still accept the new markers. More information on petri nets can be found in [Pet62].

Open workflow nets (oWFNs) are a special class of petri nets. They contain interfaces for message passing and more formalisms to model services and interaction. oWFNs can also be used to analyze the semantics of BPEL processes. By defining nets of places describing variable state like "uninitialized" or "initialized" or even more detailed, by creating state intervals such as "0–10,000", "10,001–20,000", etc. even data flow can be accurately analyzed. However, focus of recent work has been mostly with analyzing communication behavior.

[LVO+07] compares modeling concepts of petri nets for analysis of BPEL processes. [Loh07] presents a complete petri net semantics for WS-BPEL 2.0.

Petri nets have a number of drawbacks. Because of the way process state is described by a configuration of markers, places, as well as transitions and edges, the net tends to get huge in numbers. This can be reduced algorithmically after the initial model has been created, but the number is still large. The simple process shown in Chapter 5 contains:

- **As a BPEL process:** 13 activities (+ 5 implicit), 1 implicit handler (+ 3 implicit), 10 links and 4 variables
- **As a reduced petri net[1]:** 136 places, 147 transitions and 517 edges

Additionally, for the created petri net, the stemming from a particular BPEL process is barely noticeable. When a change is done in the original BPEL process (like removing an activity), the place and actual change can easily be identified. This is not the case with the corresponding petri net, which might change wildly.

[1] Command line:

```
bpel2owfn -m petrinet -f dot -p variables -p reduce MotivatingExample.bpel
>MotivatingExample_reduce.dot
```

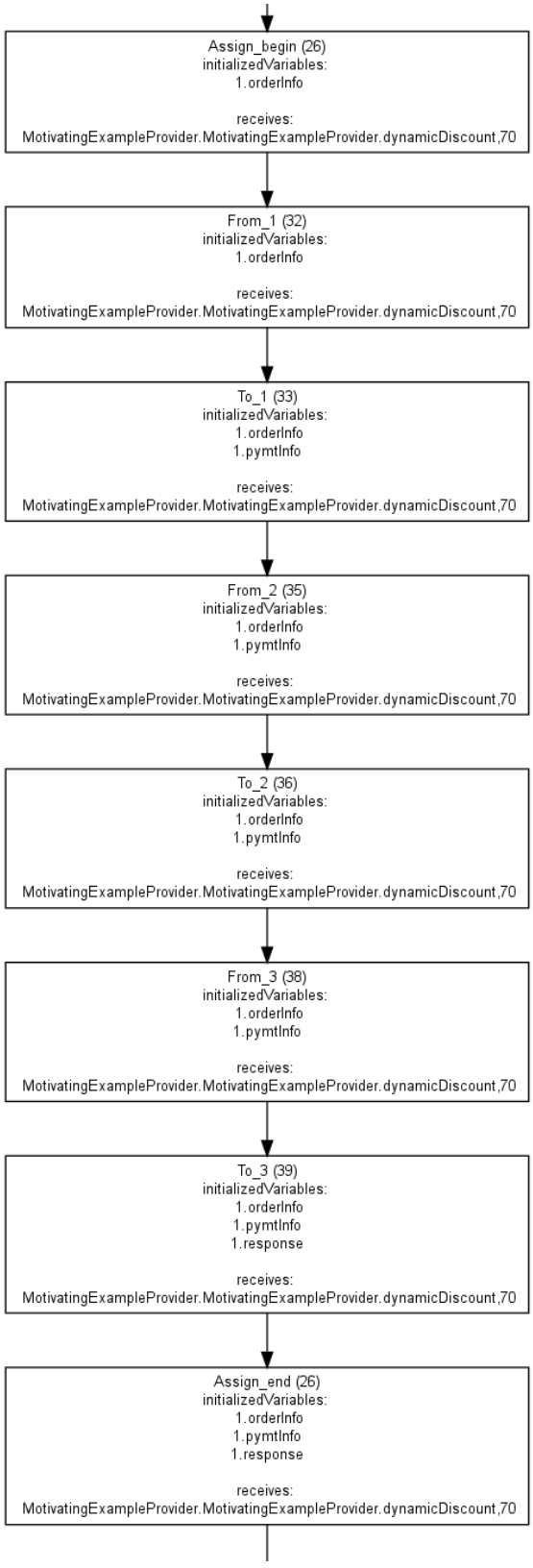


Figure 3.1: Example variable usage from BPEL2oWFN

Figure 3.1 on the preceding page shows an example of the data analysis that BPEL2oWFN provides[2]. It depicts a detail of the complete control flow graph constructed from the motivating example, also modelling BPEL variables. Visualized is the `assign` activity "B". It shows all initialized variables of each step of execution. This is the only kind of data flow analysis that BPEL2oWFN can provide at this time.

3.3 Future Work

Future work includes the comparison of the presented data flow algorithm to other data flow algorithms, as presented above. The features to be compared could be:

- Completeness concerning BPEL
- Speed or efficiency

One of the compared works could be [Loh07], which offers a complete petri net semantics for BPEL. It should be noted that BPEL2oWFN and associated tools do not yet implement any interesting solution of analyzing data flow.

Another work to compare could be [God06], which creates a CSSA representation of BPEL processes.

Additionally, [Hei03] would be an interesting candidate for comparison. It describes a way of abstractly interpreting the BPEL process for static analysis.

[2] Command line:
```
bpel2owfn -m cfg -f dot -p variables MotivatingExample.bpel >MotivatingExample.dot
```

Chapter 4

Algorithm Fundamentals

The presented data flow algorithm analyzes a BPEL process and creates data flow edges for "Def-Use" relationships. For each activity in the process, a set of possible writers is created, containing all the writers that could possibly have written to a specified variable. By examining the activities that read from variables afterwards, it is possible to create the data flow edges. The algorithm is based on the algorithm presented in [KKL07c] and [KKL08].

The presented algorithm covers the BPEL process mostly the way a BPEL navigator would traverse it. Process analysis starts at the `process` element. For container activities, analysis is started for all the directly contained activities.

When any activity analysis has finished, successors to the activity will be analyzed the way a BPEL navigator would continue: If the activity has outgoing BPEL links, the links are followed, analyzed and analysis of the target activities starts. If the activity is part of a sequence, the successor is analyzed. If the activity is contained in another activity, the container might need to do some data collecting and finish analysis.

The control flow is abstractly interpreted in places where static data flow analysis cannot achieve results that are possible with dynamic analysis.

Parallelization is handled by traversing parallel activities separately. When the parallel parts of an activity are synchronized, the separate data flows are merged.

To cover all possible execution paths through a branching activity in one analysis run, branches are analyzed separately. When the branches have been analyzed, the result of the separate analyses is merged. This is true not only for the obvious `if` and `pick` activities, but—for example—also in a `scope`'s main activity, where it is difficult to decide if the activity faults at any point, or if it continues. Also, BPEL `flow` activities may contain paths that are not executed at runtime (dead paths). These are analyzed in a special way, as described in Section 4.5 on page 31.

4.1 Variables

BPEL variables can be WSDL message types, XML Schema types or XML Schema elements. Variables are defined in the `variables` element of a BPEL `process` or `scope` activity.

Variables are accessed in two ways in BPEL. Some activities require a variable to be selected in one of their attributes. In some places, XPath queries can be used to access variables. The details on variable access can be found in [OAS07].

E^{Q_Y} is the set of all variable tuples and valid queries on each variable in the given BPEL process. An element of E^{Q_Y} is called *variable element*. $E_w^{Q_Y}$ is the set of all variable elements being written to by activities in the given BPEL process.

Complex variables contain a hierarchy of variable elements. When an activity writes to a variable element, it might write to any number of sub-elements. Again, this cannot be statically determined, so any writer to a variable element also has to be considered to be a writer to any sub-element.

4.2 Writer State

The writer state in the algorithm is saved in several writes tuples. A writes tuple is always associated to a specific activity or link and to a variable element. This is reflected in its two arguments, the first denoting the activity a or link l, the second denoting the variable element v_e. In the explanation of the algorithm, the arguments are removed, because they are needed at runtime of the analysis only.

There are three different writes tuples, $\mathsf{writes}_\circ$, $\mathsf{writes}_\bullet$ and writes_τ. $\mathsf{writes}_\circ$ is generally used as the state prior to an activity or link. $\mathsf{writes}_\bullet$ is generally used as the state after an activity or link has been executed. writes_τ is a temporary variable. In the course of analyzing a BPEL process, some of these tuples might temporarily be used differently, like the $\mathsf{writes}_\bullet$ tuple of an activity that is still being analyzed. Context and usage of a writes tuple is always important for its meaning, like it is for any variable. When a statement is true for any $\mathsf{writes}_\circ$, $\mathsf{writes}_\bullet$ or writes_τ tuple (depending on the context), "writes" is used.

A writes tuple consists of four parts. The first part is the set of possible writers. The second part is the set of disabled writers. The third part is the set of invalid writers. The fourth part is the Boolean value representing the "dead or alive" state of an activity.

The possible writers are a set containing all writing activities that could have written at the place where the writes tuple is valid. This set is mapped to poss, $\mathsf{poss}_\circ$, $\mathsf{poss}_\bullet$ or poss_τ, depending on the type of writes tuple. Formally, $\mathsf{poss} = \pi_1(\mathsf{writes})$. It is the most important set for activities, as a reading activity will read its variable from one of the writers contained therein. A writer that is contained in poss is *not required* to have written, but might be in there, because static analysis is an over-approximation.

The disabled writers are a set containing all writing activities that could have written before a possible dead path started, when the writes tuple is taken from a possible dead path. This set is mapped to dis, $\mathsf{dis}_\circ$, $\mathsf{dis}_\bullet$ or dis_τ, depending on the type of writes tuple. Formally, $\mathsf{dis} = \pi_2(\mathsf{writes})$. The set is

important in the way that it contains writers that could have written (like poss), but which are currently definitely overwritten in a dead path. After the dead path ends at some activity's join condition, these writers get moved to poss again, because the dead path might have been skipped at runtime. Writers in this set are important for the data flow algorithm itself, but not explicitly for the result. More information on dead path analysis can be found in Section 4.5 on page 31.

The invalid writers are a set containing all writing activities that have been definitely overwritten at the place where the writes tuple is valid. This set is mapped to inv, $\mathsf{inv}_\circ$, $\mathsf{inv}_\bullet$ or inv_τ, depending on the type of writes tuple. Formally, $\mathsf{inv} = \pi_3(\mathsf{writes})$. The significance of the set is minor, it is used to determine a writer duplication on multiple paths only (see Section 7.2 on page 50).

Any activity might be on a dead path, a path of linked activities starting with one activity having a `false` join condition. Thus, the activity may be dead and does not execute. The "may be dead" state—mbd for short—is a Boolean value. This state is mapped to mbd, $\mathsf{mbd}_\circ$, $\mathsf{mbd}_\bullet$ or mbd_τ, depending on the type of writes tuple. Formally, $\mathsf{mbd} = \pi_4(\mathsf{writes})$. The information conveyed in this state is the possibility of a succeeding activity executing or not executing at all, which affects dead paths. $\mathsf{mbd} =$ `false` asserts that "the following writing activity will definitely execute and overwrite everything prior to it; There is no dead path." Contrarily, $\mathsf{mbd} =$ `true` says "the following writing activity might not execute and thus not overwrite anything definitely; It might be the start of a dead path." This state is important for the data flow algorithm itself, but not explicitly for the result. More information on dead path analysis can be found in Section 4.5 on page 31.

While determining data flow, writers may temporarily appear in more than one of the writer sets. This "duplication" always happens when the data flow analysis follows more than one path after a split in the process flow and later the paths are joined. One possibility for this to happen are two paths of BPEL links in a `flow` activity. The other possibility is branching, where one of multiple branches (or paths) is taken through a branching activity, like an `if` activity. After the branching activity has been analyzed, the branches' writer state is merged and the duplication can appear.

While duplication can appear, it is not desirable. When a writer is in multiple sets, the resulting statement is not clear; The statement is equivalent to "there are multiple paths through which the state of the duplicated writer differs". This data flow algorithm also presents two solutions to the two mentioned possibilities of writer duplication. Duplication and correcting duplication is discussed in more detail in Section 7.2 on page 50.

Another important fact about writers is the handling of variable elements with respect to the whole variable. When a writer activity writes to a variable element, it is statically undecidable which parts of the variable will be overwritten. Because of that, writes to a variable (or variable element) will also be considered a write to all sub-elements. For example, if writer A writes to variable X consisting of parts $X.Y$ and $X.Z$, writer A is a writer for X, $X.Y$ and $X.Z$.

4.3 Activity and Link Flags

Analysis of a BPEL activity can happen only if all its predecessors have already been analyzed, so prior writers to an activity are known. To check if the predecessors have been analyzed, [KKL07c] and [KKL08] use a visited flag for activities and links. If the flag is `false`, it means the activity or

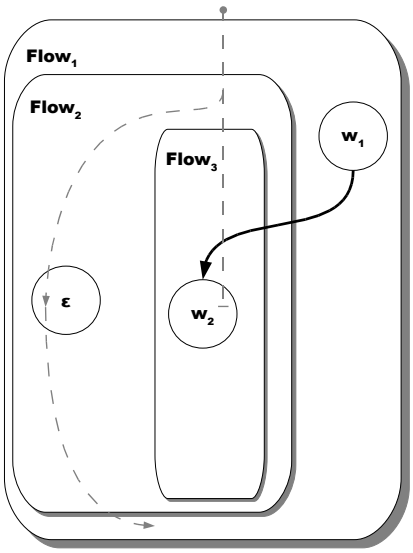


Figure 4.1: Example for failing visited flag

link has not been analyzed yet. If the flag is `true`, it means the activity or link has already been analyzed. If the flag is `true` for all activities prior of an activity, it means analysis can continue. If the flag is `false`, analysis has to stop.

An issue of the visited flag appears with the analysis of container activities in the presented algorithm. Consider a BPEL process as shown in Figure 4.1.

Analysis starts with the $\texttt{flow}_1$ activity. The only predecessor—the `process`—has its visited flag set to `true`, so visited of $\texttt{flow}_1$ is set to `true` and analysis continues with contained activity $\texttt{flow}_2$. Again, the only predecessor—$\texttt{flow}_1$—has its visited flag set to `true`, so visited($\texttt{flow}_2$) can be set to `true` and analysis continues with contained activity $\texttt{flow}_3$. The only predecessor—$\texttt{flow}_2$—has its visited flag set to `true`, so visited($\texttt{flow}_3$) can be set to `true` and analysis continues with contained activity w_2. There is an incoming link with w_2 as target and it has not been analyzed yet, so visited(w_2) stays `false` and analysis breaks here. Instead, activity ε is analyzed. Predecessor—$\texttt{flow}_2$—has visited = `true` set, so the `empty` activity is analyzed, with its visited flag set to `true`. When analysis of a contained activity has been completed, the data needs to collected. When the collecting procedure checks all activities contained in $\texttt{flow}_2$ to be visited, it yields `true`. This should imply that data collection and merging can start, because all contained activities have been analyzed. This is not true, as some contained activities have not finished analysis completely yet.

visited has been changed to flags “started” and “finished”. If an activity is in started state, analysis of the contained activities can begin. If an activity is in finished state, analysis has completely finished for it and all its contained activities. The “end of container” analysis is started only when all contained activities are in state finished. State transitions are depicted in Figure 4.2 on the facing page.

We can now go back to Figure 4.1 on the preceding page. When activity $\texttt{flow}_3$ started analysis, only the started flag has been set to `true`. After analysis of activity ε, analysis of $\texttt{flow}_2$ will not end, because $\texttt{flow}_3$ has not finished yet. When activities w_1 and w_2 have been analyzed, data from w_2 is collected, analysis of $\texttt{flow}_3$ is finished, and finished$(\texttt{flow}_3)$ is set to `true`. When data collection of activity $\texttt{flow}_2$ restarts, all contained activities will have been completely analyzed.

BPEL links do not have a started flag. When they have been analyzed, they can instantly set the finished state.

4.4 Events

Static analysis of events is a difficult process. It is statically undecidable if an event (fault, termination, `onAlarm` or `onMessage` event) occurs at a certain activity.

A fault often depends on runtime data. Termination depends on faults or ending `forEach` branches. For `forEach` branches, this is undecidable due to the static undecidability of XPath queries denoting the `forEach`'s `completionCondition`. `onAlarm` events depend on runtime data. `onMessage` events depend on external messages at runtime.

This undecidability forces the data flow algorithm to assume that a fault, event or termination handler occurs at any time. This way, the algorithm over-approximates event sources.

A problem of this undecidability is the accumulation of data flow edges in some cases. For example, `onAlarm` and `onMessage` events can happen at *any* time during the execution of a `scope` activity. More information on that particular problem can be found in Section 7.11 on page 75.

4.5 Challenges for Data Flow by DPE

Branching is a huge problem for data flow analysis in all programming languages. It is a problem because of the static undecidability of branching conditions. Static undecidability in program analysis

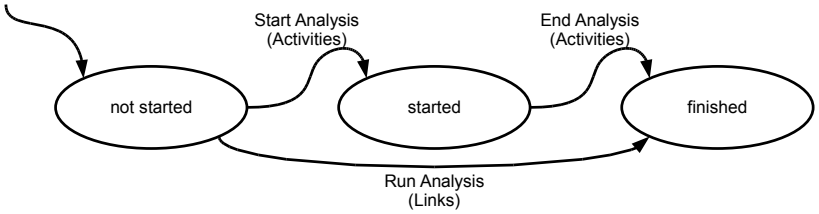


Figure 4.2: New started and visited flags

occurs in all Turing-complete computing systems (see [Tur36]). BPEL offers graph-based programming, which is novel to programming languages. Dead Path Elimination is a concept similar to branching in that the control (and data) flow splits upon a condition.

A dead path starts at a `false` join condition of an activity, regardless if the join condition itself or the incoming link status is the reason for it. A `false` join condition is the evaluation of the join condition of an activity, which references incoming BPEL links, and the evaluation yielding a `false` result. The dead path ends at a `true` join condition. A `true` join condition is the evaluation of the join condition of an activity and the evaluation yielding a `true` result. All writing activities in the dead path write only if the path is actually taken during runtime. Like the source algorithm, the presented data flow algorithm is able to determine possible writers in a graph containing dead paths, traversing the graph only once. This implies that, when following a possible dead path, current writers need to be saved. If the path is taken at runtime, the writers inside the dead path are valid and overwrite the saved writers. If the path is not taken at runtime, the saved writers are valid at the end of the dead path, when a re-enabling join condition occurs. The place to save the writers is the second writers set in the writes tuple, the dis set.

How is it being determined if a writer overwrites prior writers or—if the writer is in a dead path—only disables them? The question is answered by the Boolean value inside the writes tuple, the mbd state. If it is `false`, the next writer will invalidate all prior (that is, the possible) writers. If it is `true`, a dead path might start. The next writer will disable all prior writers only. In case the next activity is not actually a writer, this state will be passed down.

Another interesting matter is the determination of mbd. If an activity might be dead (i. e. might not execute), mbd gets `true`. If an activity will definitely execute, mbd gets `false`. The determination of mbd is done by heuristics. Influencing factors are:

- The join condition is literally `true`
- The join condition contains negations
- The actual join condition, if the aforementioned factors fail

For more information on the determination of mbd, see Section 7.2 on page 50.

As has been stated before, static data flow analysis is complicated because issues like events and branching, are undecidable. That is why the algorithm creates an over-approximation of actual possible writers.

If mbd = `false`, the next writer in the analysis will "throw away" all prior writers, by putting them into inv. This is dangerous, because the conclusion of the result is that these writers are not interesting for the current (maybe reading) activity. Thus, mbd can never be `false`, if `true` is the right value.

If mbd = `true`, the next writer in the analysis will disable prior writers only and they will be re-enabled again later. If mbd is `true` although at runtime the activity *always* executes (and thus writes), the possible writers set is too large, but not wrong. Thus, mbd can be `true`, even if `false` is the right value. The algorithm over-approximates the possible writers and the amount of times that mbd is `true` although it should be `false` is the error margin of the algorithm.

A complicated issue is the handling of dead path with respect to container activities.

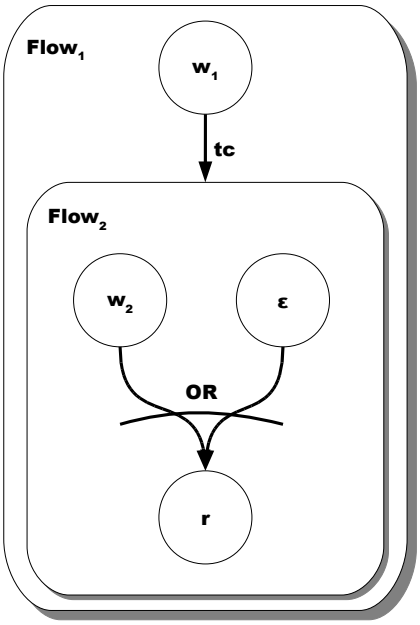


Figure 4.3: DPE issue concerning scoping

The data flow algorithm traverses the BPEL process resembling the way a BPEL navigator would traverse it. Writers need to be re-enabled (moved from dis to poss) at activities with a re-enabling join condition. A re-enabling join condition is a `true` join condition.

The join condition is determined by heuristics. If a writer is disabled on all paths joined at an activity, it needs to stay disabled. In that case, the heuristic determines the join condition to not be re-enabling. If a writer is disabled only on some paths joined, and the activity has a re-enabling join condition, the writer needs to be re-enabled.

This is not true for container activities in the original algorithm. Consider the process in Figure 4.3. At the reader, w_1 is definitely not a possible writer. However, it will be re-enabled at the join in the original algorithm. The join is a re-enabling join, because w_1 is only disabled by w_2, but still a possible writer following the `empty` activity.

To eliminate this and possibly other (yet unknown) issues with DPE and container activities, a scoping of the data flow is introduced. Disabled writers will not be taken over when starting analysis of a container activity. If a writer is disabled at the beginning of the analysis of a container activity, the container activity is still inside the dead path. This means that the disabled writer cannot be re-enabled inside the container activity. By removing the disabled writers set and then re-adding it at the end of analysis of the container activity, a correct scoping is achieved with respect to Dead Path Elimination. For more information, see Section 7.2 on page 50.

Research has been done concerning the semantics of Dead Path Elimination. [BK05] presents theoretical models—a calculus—explaining the semantics of BPEL formally. It is shown that the presence of DPE in BPEL has to be observed because of possible side effects.

[CKLW03] explains the different concepts of exception handling in graph based and structured ("algebraic") languages; DPE is seen as a way of propagating faults through a graph, much like event handlers are a way of propagating faults through structured activities.

It should be noted that the referenced work above is in both cases based on BPEL4WS 1.1.

4.6 Repeatable Activities

Repeatable activities represent another area where static data flow analysis has to overcome the issue of undecidability.

There are three kinds of loops in BPEL:

- Sequential `forEach` branches
- `repeatUntil` loops
- `while` loops

Statically, it is not possible to tell how often a loop will run, except that `repeatUntil` loops will run at least once. The solution followed in this work is to run analysis repeatedly until there is no change in the writer state. This will analyze both the effect of the writers contained inside the loop on the writers prior to the loop, as well as the effect of writers inside the loop on themselves. The analysis will not run an endless analysis loop, because the sets of the writes tuple are finite sets; At some iteration of the analysis, the sets contain at most all writers and will not change anymore, thus ending analysis of the loop. So far, no loop has been found where loop analysis did not stop after at most two iterations. However, this has not been proved yet.

4.7 Restrictions

Focus has been laid on making the algorithm as complete as possible for regular WS-BPEL 2.0. Due to time constraints on the work, few restrictions had to be placed on the improved algorithm.

There is no support for `partnerLink` analysis. This means that an `assign` writing to a `partnerLink` will not be visible in the writer state. This could be easily resolved by adding `partnerLinks` to the list of variable elements for the analysis to loop over Additionally, `assign` activities need to be checked for writes, as well as communication activities for reads.

There is no support for correlation set analysis. If any BPEL activity initiates a correlation set, this should be visible in writer state. By adding correlation sets to the list of variable elements to be looped this could be resolved Additionally, activities need to be checked for reads and writes from and to the sets.

Another important restriction is the Bernstein criterion. The Bernstein criterion states that writers may never write in parallel to a read of the same variable in processes analyzed by the presented algorithm ([Ber66]). If the Bernstein criterion holds for the process, this restriction is ensured. For a more thorough discussion on the implications of the Bernstein criterion on BPEL, see [LA94] and [KKL07b].

There is no support for "short-cut" handlers attached to `invoke` activities. These handlers are semantically equivalent to having a separate `scope` activity containing only the `invoke` activity and the specified handlers. This can be added by changing the `scope` analysis to also support `invoke` activities, to analyze the handlers.

XPath queries must return a single element only. Multi-element queries are not supported by this analysis. No solution has yet been found for properly identifying and supporting multi-element queries. One possibility for a solution is the inclusion of the work presented in [Gör08].

`rethrow` activities rethrow the original fault data, according to [OAS07]. The presented algorithm does not honor this fact. Recent writes to a fault variable are not dismissed when a `rethrow` activity is analyzed.

While event handlers may start at any time while a `scope` activity is active, they can also end at any time. The presented algorithm does not create data flow edges from event handlers back to the main activities.

When activities have been finished in analysis, the finished flag is set to `true`. This might pose a challenge when analyzing activities contained inside a loop, where analysis runs repeatedly.

It should be noted that, while most of these restrictions seem to be simple to eliminate, the implications of these changes always have to be carefully researched.

Chapter 5

Motivating Example

Figure 5.1 on the following page presents a motivating example BPEL process. It represents an order process.

A customer sends his order to start the process. A variable discount is calculated. In the end, the order is sent out, the payment is executed and the customer is notified.

Activity A is a `receive` activity. The customer's order is received and written to the `orderInfo` variable. This starts the BPEL process.

Activity B is an `assign` activity containing multiple assignments. The item price is copied from the `orderInfo` to the `pymtInfo` (payment information) variable. The account number is copied in the same way. A template response text for the customer is assigned to the `response` variable.

Next in the `flow` is a branching of activities. If the order is a "silver" order (`orderInfo.status`), activity C is executed. If the order is a "gold" order, the `scope` containing activities D, E and F is executed. If the order is neither "silver" nor "gold", execution directly continues with activity G.

Activity C presents the case that the current order is a "silver" order. The order will be rebated by 5%. Accordingly, this is added to the customer response (`response.text`). Also, the payment amount (`pymtInfo.amt`) is modified.

In the case that the current order is a "gold" order, the `scope` is executed. For "gold" orders, a variable discount is applied. The first activity to be executed is activity D.

Activity D is the invocation of a Web service. The `orderInfo` is submitted, followed by the receipt of the amount to discount (`discount` variable).

Activity E represents multiple assignments, changing both customer response (`response.text`) and payment amount (`pymtInfo.amt`) according to the received discount.

When querying for discount information of "gold" orders, an error might occur. A "gold" order, however, should not need to be paid in full in this example. That is why there is an associated fault handler containing activity F. If a fault occurs, the specified activity F is executed.

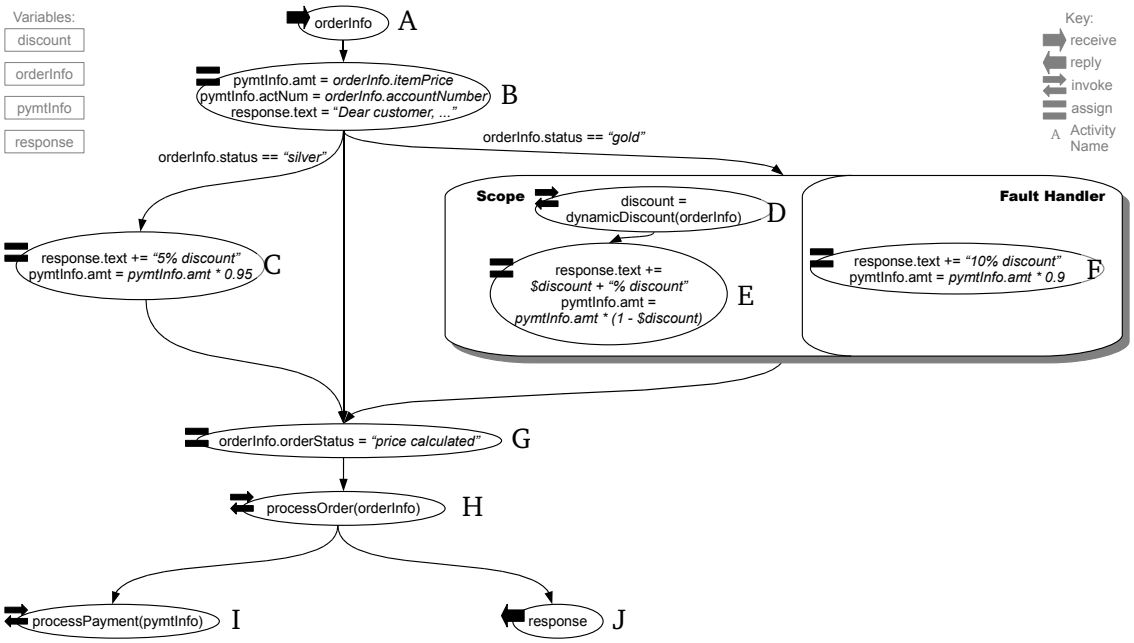


Figure 5.1: Motivating example. Different to other diagrams, activities are ellipses and contain more details. `flow` activities are not shown.

Activity F is similar to activity C. It realizes a standard discount on "gold" orders of 10%, in the case the variable discount query failed. In multiple assignments, the customer response (`response.text`) and payment information (`pymtInfo`) is modified.

Now there are several ways of reaching activity G:

1. Order is a "silver" order, 5% discount has been applied
2. Order is a "gold" order, a variable discount has been applied
3. Order is a "gold" order, but the variable discount query failed, so 10% discount has been applied
4. Order is neither "silver" nor "gold", there is no discount

Activity G confirms that the price has been successfully calculated, by assigning a text to the `orderInfo.orderStatus` variable.

Activity H is the invocation of a Web service. The `processOrder` operation is invoked and the `orderInfo` is sent to be executed. Successively, activities I and J are executed simultaneously.

Activity I is an `invoke` activity. It calls the `processPayment` operation and sends `pymtInfo`. This will execute the payment from the customer.

Activity J is a `reply` activity associated with the `receive` activity A. It sends the finished customer `response` to the customer.

With activities I and J, the example process ends.

As variables in a BPEL process are shared with all activities (excluding activities outside of the variable `scope`), there is no explicit data flow. Explicit data flow would imply messages being passed between reading and writing activities. With shared variables, if we were to partition the BPEL process and place the partitions in different BPEL engines, there would be no efficient way of distributing the variable data. The data flow algorithm presented in this work discovers the implicit data flow. The result makes partitioning and efficient distribution of the BPEL process possible, as shown in [KKL07a].

To show the applicability, the described example will be analyzed manually in Section 7.12 on page 88.

Chapter 6

Equivalence Classes

While the presented algorithm is based on the algorithm in [KKL07c], the original missed some of BPEL's features and had an issue concerning container activities. The missing features are mainly a complete analysis of the `scope` activity and other container activities. `flow`, `while` and parts of the `scope` activity have been supported. The issue with container activities was an incorrect way of reviving disabled writers.

To eliminate those deficiencies, the original algorithm had to be modified. After much work had been put into improvements of the original, the various results having different flaws, I decided to try another perspective on the problem.

The core of the algorithm lies in the various analysis methods for different activities and for BPEL links. The obvious way seemed to be to create a new grouping structure for the BPEL activities. This structure should group activities with similar behavior into the same group. Similar behavior of an activity is only the behavior regarding the order of writers, in this work. Other behavior of activities is negligible, as this work is focusing on data flow analysis. As BPEL links have no effect on the writer state of the BPEL process, there is no necessity to group them at this place. Apart from structural differences, the same analysis algorithm is to be associated with each behavioral group. This resulted in the equivalence classes presented here.

The analysis of an activity is equal to others only at the same level of detail of the data flow. Consequentially, I could create a different set of equivalence classes at each level of detail. It was discovered that a hierarchy of equivalence classes can be created, where each level of the hierarchy represented one level of detail. The most general (top) level consists of only one equivalence class, containing all BPEL activities. Starting here, the classes are again partitioned into more detailed classes. When traversing the hierarchy from top to bottom for one particular class, all features of that class regarding data flow are faced.

An activity belongs to an equivalence class exactly, if its behavior with respect to data flow and examined detail does not show a difference to the other activities in the class.

The modified algorithm presented in Chapter 7 had its most significant changes from the work put into these classes.

The top class contains all activities of BPEL.

When regarded at the lowest level of detail, all BPEL activities behave the same: Analysis has to be run on all activities. Prior writers have to be merged from incoming BPEL links, from sequential predecessors and from the hierarchical parent activity.

The class is partitioned in three subclasses: Basic activities, compensation activities and container activities.

6.1 Basic Activities

Basic activities are:

- `assign`
- `empty`
- `exit`
- `extensionActivity`
- `invoke`
- `receive`
- `reply`
- `rethrow`
- `throw`
- `validate`
- `wait`

Basic activities cannot contain any activities.

Although the `extensionActivity` might contain activities, I am putting it with basic activities. This work does not consider any extensions to standard BPEL, so the `extensionActivity` is viewed as an activity that will never read or write.

Basic activity handling is straightforward. If the activity does not write at all, nothing needs to be done. If the activity writes, it might invalidate all prior writers to the variable element. Depending on the state of $\mathsf{mbd}_\circ$ (stating if the activity might be the start of a dead path), the prior writers are either invalid or disabled. The analyzed activity is the only possible writer.

For basic activities, this covers all features relevant for data flow analysis. There is no further refining in more detailed classes.

6.2 Compensation Activities

The compensation activities `compensate` and `compensateScope` stand out next to the basic and container activities, so they form their own class.

Both do not contain any activities, but they are not basic in the sense that only the $w(a, v_e)$ status is necessary for basic activities. When a compensation activity needs to be handled, what actually is analyzed is the matching compensation handler. Matching is dependent on the place and type of compensation activity and the associated compensation handler. More information on compensation activity analysis can be found in Section 7.11 on page 75.

6.3 Container Activities

Container activities are all BPEL activities containing other activities.

Container activities are:

- `if`
- `flow`
- `forEach`
- `pick`
- `repeatUntil`
- `scope`
- `sequence`
- `while`

At a level of detail that differentiates basic and container activities only, container activities all behave the same, per definition.

Container activities can contain other activities. This means that, different to basic activities, the contained activities need to be analyzed. After all contained activities have been analyzed, the data has to be collected and merged.

Data flow analysis is “scoped” for container activities. This has been introduced to cope with issues of disabled writer reviving. Analysis of contained activities starts on a writer state that has no disabled writers and a `true` $\mathrm{mbd}_\circ$ state. At the end of the analysis, the disabled writers prior to the container activity are merged, and possibly overwritten (explained in Section 7.2 on page 50).

One additional important change to the original data flow algorithm also has to be introduced now. The nature of the scoping makes it necessary to apply additional logic to boundary-crossing links. This change obviously only affects container activities. In earlier revisions of this work, this class was partitioned into two meta-classes, one supporting boundary-crossing links and one that does not. As an algorithm has now been found that supports boundary-crossing links in all container activities, this is no longer necessary.

To go into more detail on how the container activities have to be analyzed, the following classes are "simple" and "complex" container activities, as well as `scope` activities.

It should be noted that parallel and sequential `forEach` activities behave quite differently. For that reason, the different types are in two separate classes.

6.4 Simple Container Activities

Two container activities have the simplest behavior. This class consists of:

- `if`
- Parallel `forEach`
- `pick`

These container activities behave the same way in the sense that they are "dumb" containers only with no additional logic necessary.

For the `if` activity, the contained activities are:

- **One directly contained activity:** This is the activity executed when the condition of the `if` activity is `true`.
- **One activity per `elseif`:** These activities are executed when the `if` condition is `false` and the (optional) `elseif` condition is `true`.
- **One `else` activity:** This (optional) activity gets executed when no `if` or `elseif` condition was `true`.

For the `pick` container, the contained activities are:

- **One activity per `onMessage`:** These activities run on a message receive event matching its element details.
- **One activity per `onAlarm`:** When the specified alarm fires, these activities get started.

The `forEach` activity contains exactly one `scope` activity.

For all simple container activities, it is not possible to statically determine which of the contained activities will be executed, if there is more than one. Thus, the data to be collected and merged at the end of activity analysis comes from *all* contained activities. This is different to the class of complex containers or the `scope` activity.

No further refinement needs to be done for this class.

6.5 scope Activity

The scope activity is different to the "simple" and "complex" container activities, in that the logic to analyze the data flow in scopes is more complex. As is explained in Section 7.11 on page 75 of the algorithm, scope activities are handled by first analyzing the directly contained activity, then, using the activity data flow as base, analyzing the scope handlers. At the end of the scope analysis, the data is taken from both the directly contained activity as well as the handlers.

There is no need to further refine the scope class.

6.6 Complex Container Activities

This class contains all containers with specialized logic to run contained activities and to merge contained data flow. Activities in this equivalence class are:

- Sequential forEach
- flow
- repeatUntil
- sequence
- while

What separates this class from the neighboring "simple containers" and scope class is that the algorithm part that collects and merges the contained data flow accesses the so-called "leaf activities" only. Leaf activities are the last activities to run inside the container, but not the only ones. Note that this is related to analysis only; Some of the activities in this class structurally have one contained activity only, but this activity will run multiple times. As such, there is always a designated "last activity". Control and especially data is turned over between contained activities by other mechanisms, detailed in the following sections.

This class is further refined into classes for non-repeatable and repeatable container activities.

6.7 Non-Repeatable Container Activities

This class contains all non-repeatable container activities. These are:

- flow
- sequence

These activities run only once at runtime. Data is passed between activities by merging the data of predecessors of activities. More detail is found in Section 7.2 on page 50.

This is the most detailed class for both activities.

6.8 Repeatable Container Activities

The activities in this class are repeatable activities. The class consists of:

- Sequential `forEach`
- `repeatUntil`
- `while`

Note that `forEach` is not generally a repeatable activity. In sequential mode, however, the `scope` contained inside the `forEach` activity is executed repeatedly. Sequential `forEach` activities can thus be regarded as repeatable activities.

All repeatable container activities contain exactly one activity. This contained activity is executed repeatedly. For data flow analysis, the problem of static undecidability of loop conditions appears. The presented analysis algorithm solves this by running the analysis repeatedly, until no change in writer state occurs.

This class has no further partitioning classes.

Chapter 7

The Algorithm

In this chapter I will present the data flow algorithm. The algorithm is based on [KKL07c] and [KKL08], but has been improved. The original algorithm had shortcomings and left out details of BPEL semantics by choice. In more detail:

- Issues with disabled writers in container activities
- Issues with writer duplication
- No support for container activities, except for `flow`, `scope` and `while`
- No complete support for `scope` activities and all its handlers
- Issues regarding activity preconditions
- Issues with cross-boundary BPEL links

These issues have been resolved. The thesis presents an algorithm supporting the whole WS-BPEL 2.0 semantics including event, fault, termination and compensation handlers.

The following chapters will provide details of the different parts of the data flow algorithm. In the process of doing this, I am also going to repeat some of the explanations of the algorithm which have already been made in the referenced sources. It should thus be possible to understand the presented algorithm without reading any references.

The symbol notations used in the presented algorithm and their meanings are listed in Table A.1 on page 101.

The following sections cover these topics:

- **Section 7.1 "General Activity Handling"** provides an introduction and Algorithms 1 "HANDLEACTIVITY" and 2 "HANDLESUCCESSORS"
- **Section 7.2 "Data Handling"** explains the handling of writer states and provides the Algorithms 3 "MERGEDATA" and 4 "overWrite"
- **Section 7.3 "Begin and End of Analysis"** presents the start and end of process analysis
- **Section 7.4 "Basic Activities"** to **Section 7.11 "`scope` Activity"** presents analysis algorithms for various BPEL features and explains their details and interaction

- **Section 7.12 "Revisiting the initial example"** executes the algorithm on the example provided in Chapter 5
- **Section 7.13 "Conclusion"** concludes the algorithm

7.1 General Activity Handling

Analysis of an activity is performed by the following steps:

1. **Check preconditions:** Check whether the activity can actually run based on incoming links, sequential predecessors and its parent
2. **Merge data:** Merge the data of all predecessors (links, sequence, parent) into $\mathsf{writes}_{\circ}$
3. **Core handling:** After determining the type of activity, run specialized handler procedures
4. **Successors:** After the activity has been analyzed, HANDLESUCCESSORS is executed. This examines outgoing links, sequential successors and—if the activity was the last to run in a container activity—the end of the parent activity.

These steps are realized in the procedures HANDLEACTIVITY and HANDLESUCCESSORS.

The procedure HANDLEACTIVITY is described in Algorithm 1 on the next page.

Generally, this procedure will take an activity (a) and a variable element (v_e) as arguments. Each activity has multiple incoming data flows: (i) from its parent (ii) from its incoming links and (iii) from the sequential processing it might be in. First, it is determined whether all these possible predecessors have already been finished. This is done in lines 4 to 7.

If not, we do not need to go on handling the activity. Control links cannot form cycles, so it is impossible to have activities that never have their preconditions fulfilled. HANDLEACTIVITY will be called again when all predecessors have been handled.

If the predecessors have been finished, we have all the data needed to process the activity. The call to MERGEDATA (line 14) merges the writer states prior to the activity into $\mathsf{writes}_{\circ}$ (see Section 7.2 on page 50). After the call to MERGEDATA, $\mathsf{writes}_{\circ}$ contains all possible, disabled and invalid writers that ran before a. Additionally, the "may be dead" state of the activity has been determined.

The next part determines the type of activity that is currently being handled and delegates the handling to designated procedures (line 18-41). This is the main part of HANDLEACTIVITY. Each procedure calls HANDLESUCCESSORS at the end of the analysis. HANDLESUCCESSORS continues traversing the process model according to the abstract execution semantics. HANDLESUCCESSORS is presented in Algorithm 2 on page 51.

The following points have to be made about core handling: If we are analyzing the `process` itself, HANDLEACTIVITY will continue with the directly contained activity. Since the preconditions for the `process` are always fulfilled, this call to HANDLEACTIVITY will always return after completing the analysis. We can copy the final $\mathsf{writes}_{\bullet}$ tuple (line 27-28). In a `sequence` activity, the only activity without a sequential predecessor is analyzed (line 29-31). For `scope` activities, analysis is started for the directly contained activity only (line 23). `scope` handlers need the result of this

Algorithm 1 Handling of an activity

procedure HANDLEACTIVITY(a,v_e)

  // Preconditions
  parentHandled $\leftarrow a = \mathsf{process} \vee \mathsf{started}(p), (p,a) \in \mathsf{HR}$
  allLinksHandled $\leftarrow \mathscr{L}_{in}(a) = \emptyset \vee \forall l \in \mathscr{L}_{in}(a) : \mathsf{finished}(l)$
  sequenceHandled $\leftarrow (p,a) \notin \mathsf{SR} \vee \mathsf{finished}(p), (p,a) \in \mathsf{SR}$
  if *parentHandled* $\wedge$ *allLinksHandled* $\wedge$ *sequenceHandled* **then**
    $\mathsf{started}(a) \leftarrow \mathtt{true}$

    // Update $\mathsf{writes}_{\circ}(a, v_e)$
    if $a = \mathsf{process}$ **then**
      $\mathsf{writes}_{\circ}(a, v_e) \leftarrow (\emptyset, \emptyset, \emptyset, \mathtt{false})$
    else
      MERGEDATA(a, v_e)
    end if

    // Core handling
    if $a \in \mathscr{A}_{basic}$ **then**
      HANDLEBASICACTIVITY(a,v_e)
    else if $a \in \mathscr{A}_{flow}$ **then**
      HANDLEFLOW(a,v_e)
    else if $a \in \mathscr{A}_{scope}$ **then**
      HANDLEACTIVITY(sa, v_e), $(a, sa) \in \mathsf{HR}$, $sa \in \mathscr{A}$
    else if $a \in \mathscr{A}_{while} \cup \mathscr{A}_{repeatuntil}$ **then**
      HANDLEACTIVITY(n, v_e), $(a, n) \in \mathsf{HR}$
    else if $a = \mathsf{process}$ **then**
      HANDLEACTIVITY(a',v_e), $(a,a') \in \mathsf{HR}$
      $\mathsf{writes}_{\bullet}(a, v_e) \leftarrow \mathsf{writes}_{\bullet}(a', v_e)$
    else if $a \in \mathscr{A}_{seq}$ **then**
      $\exists a_{nest}, (a, a_{nest}) \in \mathsf{HR} : \forall a_{pred}, (a, a_{pred}) \in \mathsf{HR} : (a_{pred}, a_{nest}) \notin \mathsf{SR}$
      HANDLEACTIVITY(a_{nest}, v_e)
    else if $a \in \mathscr{A}_{comp}$ **then**
      HANDLECOMPENSATE(a, v_e)
    else if $a \in \mathscr{A}_{pick}$ **then**
      HANDLEPICK(a, v_e)
    else if $a \in \mathscr{A}_{foreach}$ **then**
      $s : (a,s) \in \mathscr{A}_{scope}$
      HANDLEACTIVITY(s, v_e)
    else if $a \in \mathscr{A}_{if}$ **then**
      HANDLEIF(a, v_e)
    end if
  end if
end procedure

analysis as input, so they cannot be started immediately. In the analysis of the `scope` activity, HANDLESCOPEHANDLERS is called, which in turn analyzes the handlers.

This concept is also used for `while` and `repeatUntil` loops (line 25). HANDLEENDOFLOOP checks whether the loop has to be analyzed again or if HANDLESUCCESSORS may be called (see Section 7.8 on page 70). Other activities have specialized handlers that will be explained later.

All of these "next step" activities will quit fast (if needed) and do a fragmented *depth first search*. If the preconditions are not satisfied (see above: parent, linked source or sequential predecessor not handled), analysis of those activities will immediately fail and return. They will be reached and handled through another path. If the activity can be handled (preconditions satisfied), it will only return after a maximum depth has been reached. The maximum being the depth of an activity that has unsatisfied preconditions.

HANDLESUCCESSORS (Algorithm 2 on the next page) determines and executes the next step of the analysis. If we have outgoing links, those are handled (line 2-4). If the activity is part of a sequence, the successor is handled (line 5-7). If the handled activity was part of a container activity and was the last to be run (determined in the HANDLEENDOF* procedures described later), the "end-of-container" procedure is executed (line 8-32).

The following things should be noted for successor handling: HANDLEENDOFFLOW (line 11) will be called if the caller of HANDLESUCCESSORS is one of the "leaf activities" only (see Section 7.6 on page 67). A "leaf activity" is an activity directly contained in a `flow` activity, without any outgoing links to descendants of the `flow`. HANDLEENDOFSEQUENCE will be called if the caller has no sequential successors (line 13). If the caller was the directly contained activity of a `scope` activity, all the writers needed for the handlers are present. Then, analysis will continue with the event, fault and termination handlers of the `scope` (see Section 7.11 on page 75) as seen in line 15. In turn, if one of the handlers calls HANDLESUCCESSORS, HANDLEENDOFSCOPE can finally run (line 17).

For `forEach` activities (line 22-27), the successors depend on its type: If it is a parallel `forEach`, then all threads running in parallel have the same analysis result and the loop can be ended. If the `forEach` was sequential, we invoke the end of a loop activity (HANDLEENDOFLOOP). If a fix-point is not reached, it causes an additional analysis run on the `forEach` activity.

Activities in an `if` activity are distributed in two levels of hierarchy. An activity (condition is `true`) is always contained directly in the `if` activity. Other activities (condition is `false`) are contained in `elseif` and `else` containers. To check whether the ended activity is contained inside an `if` activity, we need to check both the parent and the grandparent. If one of them is an `if` activity, HANDLEENDOFIF can be executed. `if` activities are handled in line 29-31.

7.2 Data Handling

For the analysis of a BPEL process, two algorithms are needed for general data handling. MERGEDATA and overWrite[1] are presented in this section.

[1] *Note:* Unlike other algorithms, the name "overWrite" is not written in small capitals. Small capitals are used for procedures only, functions are written in regular typeface.

Algorithm 2 Handling of activity successors

```
1:  procedure HANDLESUCCESSORS(a, v_e)
2:      for all l ∈ 𝒟_out(a) do
3:          HANDLELINK(l,v_e)
4:      end for
5:      if (a,s) ∈ SR then
6:          HANDLEACTIVITY(s,v_e)
7:      end if
8:      p : (p, a) ∈ HR
9:      pp : (pp, p) ∈ HR
10:     if p ∈ 𝒜_flow ∧ ¬∃l : (a,l,a') ∈ LR, a' ∈ descendants(p) then
11:         HANDLEENDOFFLOW(p, v_e)
12:     else if p ∈ 𝒜_seq ∧ (a, a') ∉ SR then
13:         HANDLEENDOFSEQUENCE(p, v_e)
14:     else if p ∈ 𝒜_scope then
15:         HANDLESCOPEHANDLERS(p, v_e)
16:     else if p ∈ ℋ_event ∨ p ∈ ℋ_fault ∨ p ∈ ℋ_term then
17:         HANDLEENDOFSCOPE(pp, v_e)
18:     else if p ∈ 𝒜_while ∪ 𝒜_repeatuntil then
19:         HANDLEENDOFLOOP(p, v_e)
20:     else if p ∈ ℋ_pick then
21:         HANDLEENDOFPICK(pp, v_e)
22:     else if p ∈ 𝒜_foreach then
23:         if parallel(p) = yes then
24:             writes•(p, v_e) ← writes•(a, v_e)
25:         else
26:             HANDLEENDOFLOOP(p, v_e)
27:         end if
28:     else if p ∈ 𝒜_if then
29:         HANDLEENDOFIF(p, v_e)
30:     else if pp ∈ 𝒜_if ∧ p ∉ 𝒜 then
31:         HANDLEENDOFIF(pp, v_e)
32:     end if
33: end procedure
```

The procedure MERGEDATA merges the data from all possible predecessors of an activity. The function overWrite takes two writes tuples and returns a new writes tuple. The input—$\mathsf{writes}_\circ$ and writes_τ—is the the state of writers before an activity ran and the state of writers at any exit point, respectively. An exit point may be the end of a basic activity or the end of a container activity. The function determines the new state of writers after the exit point and return it to the caller.

Algorithm 3 Data merging

1: **procedure** MERGEDATA(a, v_e)
2:
3: // Parent or predecessor
4: **if** $(a_{pred}, a) \notin \mathsf{SR}$ **then**
5: $Let(a_{parent}, a) \in \mathsf{HR}$
6: $\mathsf{writes}_\circ(a, v_e) \leftarrow (\mathsf{poss}_\circ(a_{parent}, v_e), \emptyset, \mathsf{inv}_\circ(a_{parent}, v_e), \texttt{false})$
7: **else**
8: $\mathsf{writes}_\circ(a, v_e) \leftarrow (\mathsf{poss}_\bullet(a_{pred}, v_e) \cup \mathsf{dis}_\bullet(a_{pred}, v_e), \emptyset, \mathsf{inv}_\bullet(a_{pred}, v_e), \texttt{false})$
9: **end if**
10:
11: // Links
12: **if** $(|\mathscr{L}_{in}(a)| = 1 \wedge \neg\mathsf{negateslinkstatus}(\mathsf{jc}(a))) \vee \mathsf{jc}(a)$ is a logical AND over all incoming links **then**
13: $P \leftarrow \bigcup_{l \in \mathscr{L}_{in}(a)} \mathsf{poss}_\bullet(l, v_e)$
14: $D \leftarrow \bigcup_{l \in \mathscr{L}_{in}(a)} \mathsf{dis}_\bullet(l, v_e)$
15: **else**
16: $P \leftarrow \bigcup_{l \in \mathscr{L}_{in}(a)} \mathsf{poss}_\bullet(l, v_e) \cup \mathsf{dis}_\bullet(l, v_e) \setminus \bigcap_{l \in \mathscr{L}_{in}(a)} \mathsf{dis}_\bullet(l, v_e)$
17: $D \leftarrow \bigcap_{l \in \mathscr{L}_{in}(a)} \mathsf{dis}_\bullet(l, v_e)$
18: **end if**
19: $I \leftarrow \bigcup_{l \in \mathscr{L}_{in}(a)} \mathsf{inv}_\bullet(l, v_e)$
20:
21: // Determine $\mathsf{mbd}_\circ(a, v_e)$
22: $d \leftarrow \begin{cases} \texttt{false} & \mathsf{alwaystrue}(\mathsf{jc}(a)) \\ \texttt{true} & \mathsf{negations}(\mathsf{jc}(a)) \\ \neg[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet} & \text{otherwise} \end{cases}$
23: $\mathsf{writes}_\circ(a, v_e) \leftarrow \mathsf{writes}_\circ(a, v_e) \sqcup (P, D, I, d)$
24: PURGEDUPLICATESOR($\mathsf{writes}_\circ$)
25: **end procedure**

The MERGEDATA procedure is presented in Algorithm 3. The possible, disabled and invalid writers of the parent, sequential predecessor and all incoming links are collected and merged, doing special handling along the way.

Line 6 shows data collection if the activity is not part of a sequence or the beginning of a sequence. The data is pulled from its parent.

One thing to note is the emptying of the disabled writer set and the reset of $\mathtt{mbd}_\circ$ to `false`. This is a design decision of the algorithm. If the activity is the first to run in a container, the disabled writers need to be "scoped" first. "Scoping" concerning the topic of data flow means that the inside of a container is analyzed separately from the outside. The scoping is necessary, because disabled writers from the outside of a container activity cannot be re-enabled inside the container activity.

If a writer is disabled at an activity, it means that the activity is inside a dead path that started after the disabled writer. Re-enabling means that the dead path has ended. Usually, disabled writers are re-enabled when the data flow comes along an enabling join condition. BPEL semantics dictate that the dead path encloses the whole container activity. When all activities inside a container activity lie in the dead path, disabled writers cannot be re-enabled.

The scoping mechanism clears the disabled writers when analysis of a container activity starts. At the end of the analysis of the container activity, the former disabled writers are merged in again. This way, disabled writers from outside the container activity will never be re-enabled inside the container activity. The merging is done in function overWrite.

$\mathtt{mbd}_\circ$ of the container activity will never be inherited by the first activity in a container. The data flow is scoped, the inside and outside of a container activity analyzed separately. If the container activity had a $\mathtt{mbd}_\circ$ status of `true`, that is a concern for the overWrite function after the analysis of the container activity ended, but does not transfer to this activity. If the first activity might be dead because of its join condition, that will be determined later in the algorithm, when links are analyzed. Thus, $\mathtt{mbd}_\circ$ is set to `false`. Figure 7.1 on the next page is a graphical representation of these steps.

Line 8 shows data collection if the activity is part of a sequence, but not the beginning of it. The $\mathtt{writes}_\circ$ information is taken from the preceding activity.

A dead path does not propagate through a sequence. If an activity of a sequence has a `false` join condition, it will not run. The succeeding activity in the sequence does not lie inside the dead path of the preceding activity. Its execution depends on the parent and on its incoming links only, not on its predecessor. Thus, disabled writers of preceding activities can be re-enabled and $\mathtt{mbd}_\circ$ can be set to `false`. Figure 7.2 on page 55 exemplifies how this is done.

Lines 12 to 19 handle the writers of incoming links inside `flow` activities. The branches of the `if` statement determine whether disabled writers get re-enabled at the current activity. If there is one incoming link only and the activity's join condition does not negate the link status, or if the join condition is a logical *AND* over the incoming links, disabled writers do not get re-enabled. Therefore, they are copied from the link's $\mathtt{writes}_\bullet$ tuple (line 13-14).

If the only incoming link negates the link status or if the join condition is a logical *OR* over the incoming links, the dead path ends at this point. The dead path is only ending in the abstract interpretation of the BPEL process. As the abstract interpretation presents an over-approximation, the dead path might already have ended. At the end of the dead path, disabled writers are re-enabled (line 16-17). When a writer is disabled on all incoming links, this means that the dead path does not end for this particular writer. It stays disabled.

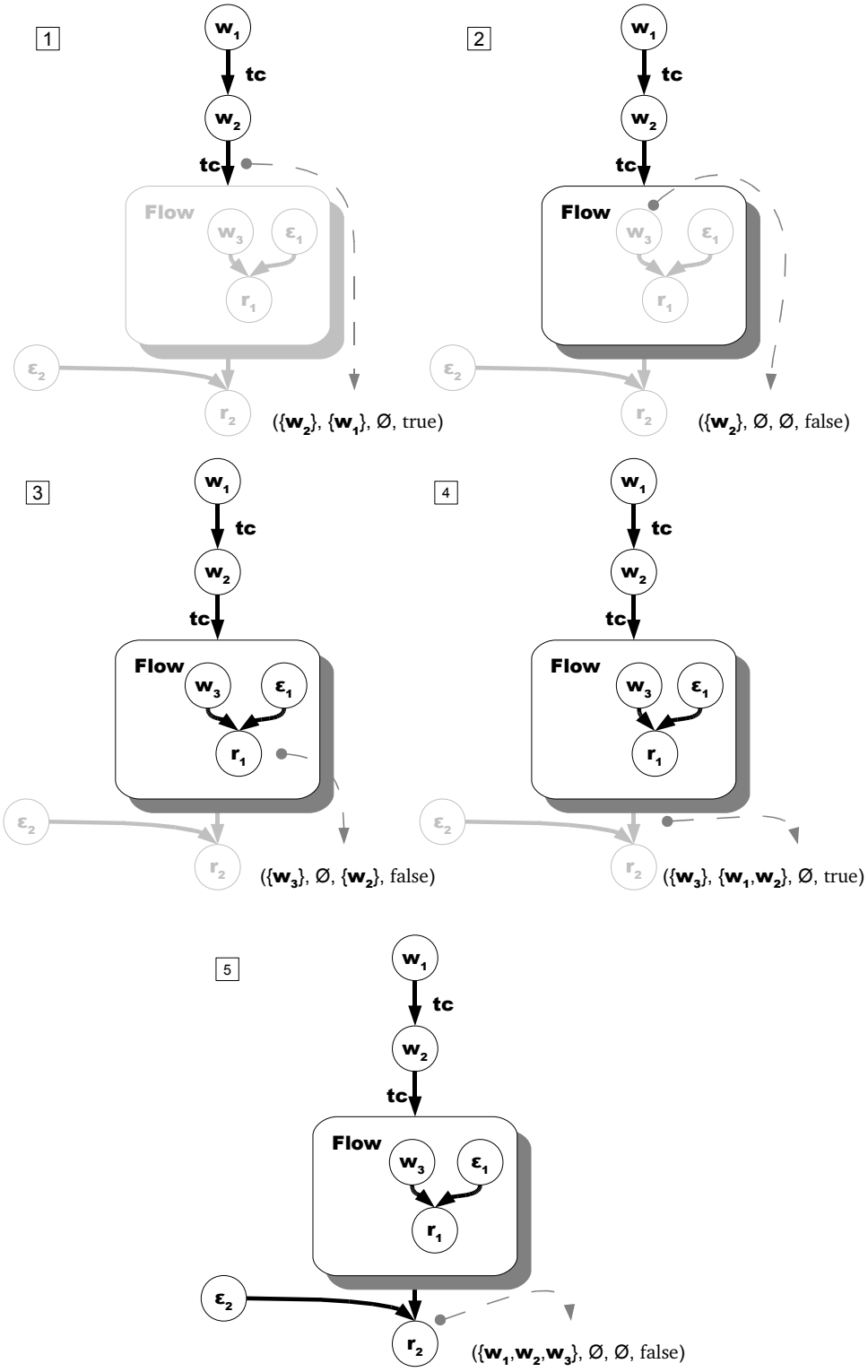


Figure 7.1: Data flow scoping

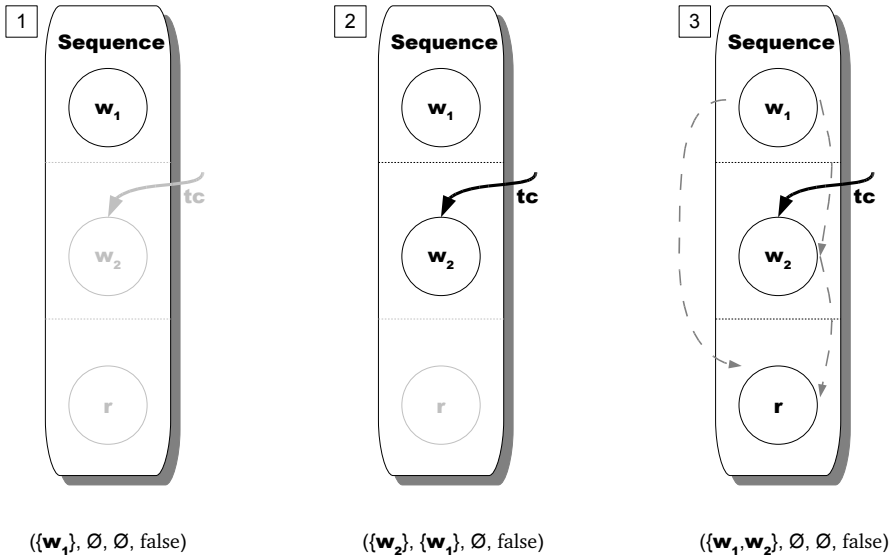


Figure 7.2: Re-enabling writers in a sequence

After possible and disabled writers have been determined, the invalid writers are copied from the incoming links as well (line 19).

In line 22, the $\mathsf{mbd}_\circ$ state of the activity is computed.

- **If the join condition is always `true`, $\mathsf{mbd}_\circ$ gets `false`.** The join condition determines the $\mathsf{mbd}_\circ$ state of an activity. Even if the activity lay inside a dead path, a `true` join condition always enables it.
- **If the join condition contains negations, $\mathsf{mbd}_\circ$ gets `true`.** The only incorrect situation for the data flow analysis is that we put possible writers in $\mathsf{inv}_\circ$ when we should not. This happens if $\mathsf{mbd}_\circ$ is `false` even though the correct value should be `true`. The problem with putting writers into $\mathsf{inv}_\circ$, when they might need to be re-enabled, is that the algorithm would make an incorrect statement about possible writers. Variables written to by an invalid writer can never be re-enabled and are "lost". On the other hand, putting a writer into $\mathsf{dis}_\circ$ when it is not necessary to ever re-enable it is an acceptable over-approximation. Thus, a one-sided error is acceptable ($\mathsf{mbd}_\circ$ can be `true` even if it the correct value is `false`). This way, writers that could be re-enabled will never be put into $\mathsf{inv}_\circ$. We cannot accept an error towards $\mathsf{mbd}_\circ = \texttt{false}$, but cannot statically decide the result of the join condition, so $\mathsf{mbd}_\circ$ is set to `true`.
- **In other cases, abstractly interpret the join condition.** $\neg[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$ denotes that we evaluate the join condition (jc) based on the activity (a) and variable element (v_e) using the negation of the $\mathsf{mbd}_\bullet$ matching the incoming links ($\neg\mathsf{mbd}_\bullet$). The result is negated again ($\neg[\![a, v_e]\!]^{jc}_{\neg\mathsf{mbd}_\bullet}$). This is semantically equivalent to evaluating link states in the join condition for the case that most incoming links are dead and using the result as condition for the execution of the activity.

As an simple illustration, consider the links l_1, l_2, l_3, $\mathsf{mbd}_\bullet(l_1, v_e) = \texttt{true}$, $\mathsf{mbd}_\bullet(l_2, v_e) = \texttt{true}$, $\mathsf{mbd}_\bullet(l_3, v_e) = \texttt{false}$. Note that $\mathsf{mbd}_\bullet$ of the links is already an over-approximation of their status: $\mathsf{mbd}_\bullet$ might be `true`, even if the link status is always `true` at runtime. If the join condition is $(l_1 \wedge l_2) \vee l_3$, the evaluation of the join condition yields a `true` result. Negating the join condition, the final result is $d \leftarrow \texttt{false}$.

Finally, writer states from either sequence or hierarchy and incoming links is combined (line 23).

Because of duplicates introduced with the joining of paths having their source anywhere in the same writer, line 24 calls PURGEDUPLICATESOR.

Algorithm 4 Data overwriting

```
1:  function overWrite(writes_○, writes_τ)
2:      if poss_τ = ∅ then
3:          writes_• ← writes_○
4:      else if poss_τ ≠ ∅ ∧ mbd_○ then
5:          writes_• ← (poss_τ ∪ dis_τ, dis_○ ∪ poss_○, inv_○ ∪ inv_τ, poss_○ ∩ (poss_τ ∪ dis_τ) ≠ ∅)
6:      else if poss_τ ≠ ∅ ∧ ¬mbd_○ then
7:          writes_• ← (poss_τ ∪ dis_τ, dis_○, inv_○ ∪ inv_τ ∪ poss_○, false)
8:      end if
9:      PURGEDUPLICATESXOR(writes_•)
10:     return writes_•
11: end function
```

In Algorithm 4, the function overWrite is presented. Its arguments are two writes tuples, $\mathsf{writes}_\circ$ and writes_τ and it returns a new $\mathsf{writes}_\bullet$ tuple. overWrite will take $\mathsf{writes}_\circ$ as an earlier state, the state of writers before a container activity or a basic activity. This state will be updated with the state of writes_τ. writes_τ can be one of the following:

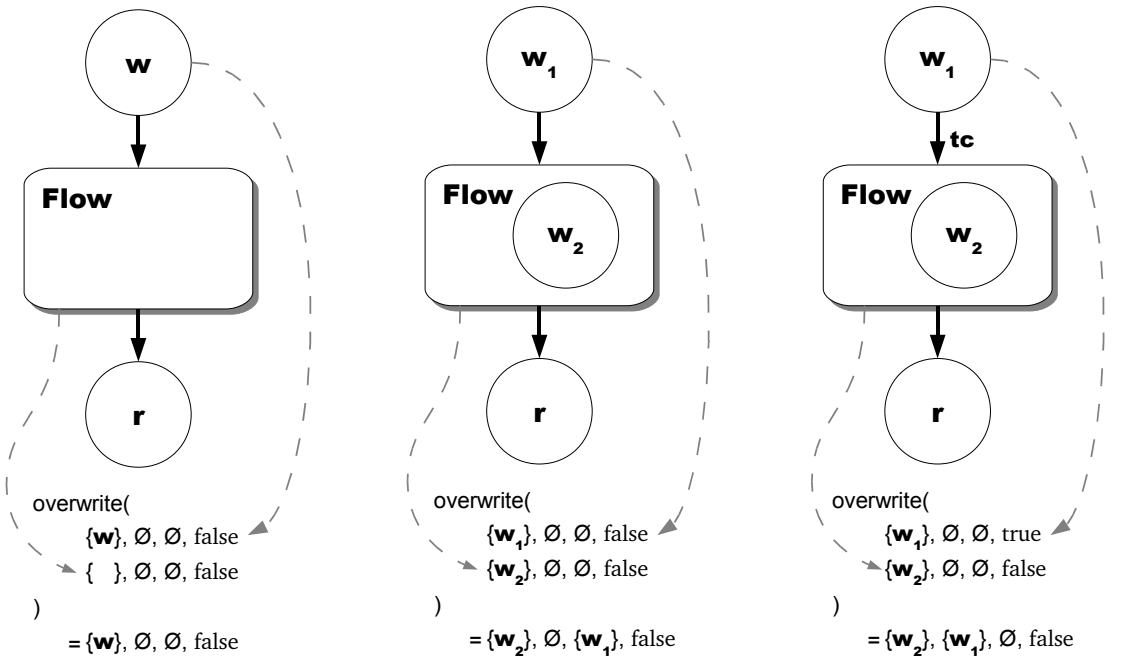
- The writers at the end of analysis of the inside of a container up to a certain point, or
- An empty tuple with one single possible writer for a basic activity.

This effectively updates the writer states. The data flow analysis is "scoped", as described before. Especially regarding activities having $\mathsf{mbd}_\circ = \texttt{true}$, this scoping is an important factor for this algorithm. The state of $\mathsf{mbd}_\circ$ determines how writes_τ will update $\mathsf{writes}_\circ$. writes_τ cannot be used "as is" after a container activity without checking $\mathsf{writes}_\circ$.

The function works for updating the writer states after basic activities (Section 7.4 on page 64) and at the end of containers (see Section 7.5 on page 65). Generally, overWrite will merge the tuples, but there are challenges, as described below. Figure 7.3 on the facing page is a graphical representation of the most basic cases that overWrite is used in.

The merging is done in lines 2-8 and resembles the way basic activities are handled (see Section 7.4 on page 64). In fact, basic activities are handled the same way as in the cited technical report by putting the basic activity as only possible writer into writes_τ (more in Section 7.4 on page 64).

- writes_τ **contains no new writers (line 3):** In that case, there is no need to update. All states stay as they were in $\mathsf{writes}_\circ$.



(a) Empty container activity

(b) Contained activity, default transition condition

(c) Contained activity, non-default transition condition

Figure 7.3: Basic examples of function overWrite

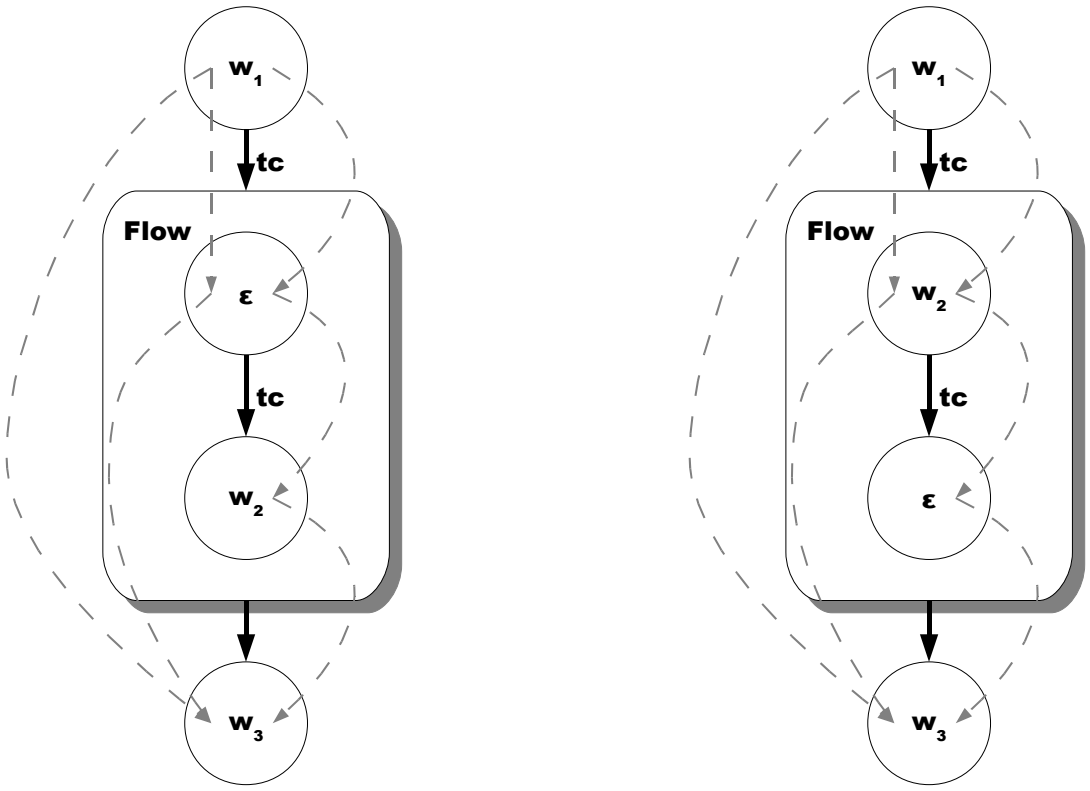
- writes_τ **contains new possible writers. The new writers might not execute due to DPE ($\mathsf{mbd}_\circ = \mathtt{true}$, line 5):**

  The writers in poss_τ have written after the writers in $\mathsf{poss}_\circ$. They are put into $\mathsf{poss}_\bullet$.

  The writers in dis_τ have been disabled somewhere inside the container activity. Basic activities never set dis_τ, so this applies only to container activities. As the disabled writers set is cleared before entering a container activity, dis_τ contains disabled writers from inside the container activity only, none from outside. The dead path that started after the disabled writer ends when calling overWrite, as it cannot continue after the end of the container. There is no way to statically decide if the dead path has been executed or not. That is why the disabled writers from dis_τ now have to be re-enabled. They are put into $\mathsf{poss}_\bullet$.

  The writers in $\mathsf{dis}_\circ$ are writers that have been disabled before the activity started. These writers cannot be re-enabled inside the activity. They are put into $\mathsf{dis}_\bullet$ and thus stay disabled.

  The writers in $\mathsf{poss}_\circ$ are overwritten by the activity. As $\mathsf{mbd}_\circ = \mathtt{true}$, the whole activity might not execute. The activity is the first writing basic activity or the first container activity containing writers in a dead path. The writers from $\mathsf{poss}_\circ$ need to be disabled. The writers from $\mathsf{poss}_\circ$ are put into $\mathsf{dis}_\bullet$.



(a) One path has writers, one does not

(b) Both paths have writers

Figure 7.4: mbd propagation through a container activity that has two data flow paths

The writers from $\mathsf{inv}_\circ$ are the invalid writers from before the activity. The writers from inv_τ are the invalid writers from inside the activity, if this is a container activity. For basic activities, $\mathsf{inv}_\tau = \emptyset$. Both $\mathsf{inv}_\circ$ as well as inv_τ are put into $\mathsf{inv}_\bullet$.

If the activity was a container activity, there might be data paths through it that do not contain writers, and paths that contain writers. In these cases, determination of $\mathsf{mbd}_\bullet$ is complex. $\mathsf{mbd}_\circ$ can always simply be copied to $\mathsf{mbd}_\bullet$. This covers the case shown in Figure 7.4(a). There are multiple data flow paths through the `flow` activity, and only some contain writing activities. When $\mathsf{mbd}_\circ$ is copied to $\mathsf{mbd}_\bullet$, w_3 disables, but does not invalidate w_1. The case shown in Figure 7.4(b) is covered as well. There are again multiple data flow paths through the `flow` activity, but all contain writing activities. When $\mathsf{mbd}_\circ$ is copied to $\mathsf{mbd}_\bullet$, w_3 again disables, but does not invalidate w_1. This is acceptable. To improve the data flow algorithm, $\mathsf{mbd}_\bullet$ can be made dependent on the actual data flow paths. poss_τ contains the most current writers. If there were data flow paths through the activity that did not have writers, then poss_τ and $\mathsf{poss}_\circ$ overlap. The overlapping area contains the writers from $\mathsf{poss}_\circ$, because these were not overwritten. The non-overlapping part contains writers from other data flow paths through the container activity.

If $\mathsf{poss}_\circ$ and poss_τ overlap ($\mathsf{poss}_\circ \cap \mathsf{poss}_\tau$), the writers from before the activity started are still valid, and so $\mathsf{mbd}_\bullet$ can be set to `true`. This means the next activity will still only disable the writers of $\mathsf{poss}_\bullet$, not invalidate them. The important change is that when poss_τ and $\mathsf{poss}_\circ$ *do not* overlap, then there is no data flow path through the activity that does not write, and $\mathsf{mbd}_\bullet$ can be set to `false`. This improves the mbd propagation slightly, because subsequent activities can now safely invalidate the writers in $\mathsf{poss}_\bullet$.

The same reasoning can be used for dis_τ. dis_τ contains writers from outside the container activity and from the inside that were overwritten in a dead path inside the container activity. If dis_τ contains writers from outside the container activity, then there is a data flow path through the container activity that leaves the writers of $\mathsf{poss}_\circ$ valid. Again, if dis_τ overlaps with $\mathsf{poss}_\circ$, $\mathsf{mbd}_\bullet$ can be set to `true`; And more importantly, when dis_τ does not overlap with $\mathsf{poss}_\circ$, $\mathsf{mbd}_\bullet$ can be set to `false`.

To cover both poss_τ as well as dis_τ, they are merged, and then checked for overlapping with $\mathsf{poss}_\circ$: $\mathsf{poss}_\circ \cap (\mathsf{poss}_\tau \cup \mathsf{dis}_\tau)$.

Note that this concept also covers the case of poss_τ not containing any writers. To keep the algorithm reasonably simple, that case has been kept separately, as in the original algorithm for basic activity writers.

- writes_τ **contains writers and the contained writers will always execute** ($\mathsf{mbd}_\circ =$ `false`**, line 7):** Old writers are put into $\mathsf{inv}_\bullet$. New writers are put into $\mathsf{poss}_\bullet$. The disabled writers contained in the "update" are merged into $\mathsf{poss}_\bullet$ (see previous paragraph for an explanation). $\mathsf{mbd}_\circ$ is kept `false`.

Because of duplicates introduced with the "update" of writers already in $\mathsf{writes}_\circ$ with information from writes_τ, line 9 calls PURGEDUPLICATESXOR.

In the end (line 10), the $\mathsf{writes}_\bullet$ tuple is returned to the caller. $\mathsf{writes}_\bullet$ contains the state of writers after $\mathsf{writes}_\circ$ got updated with the scoped information from writes_τ.

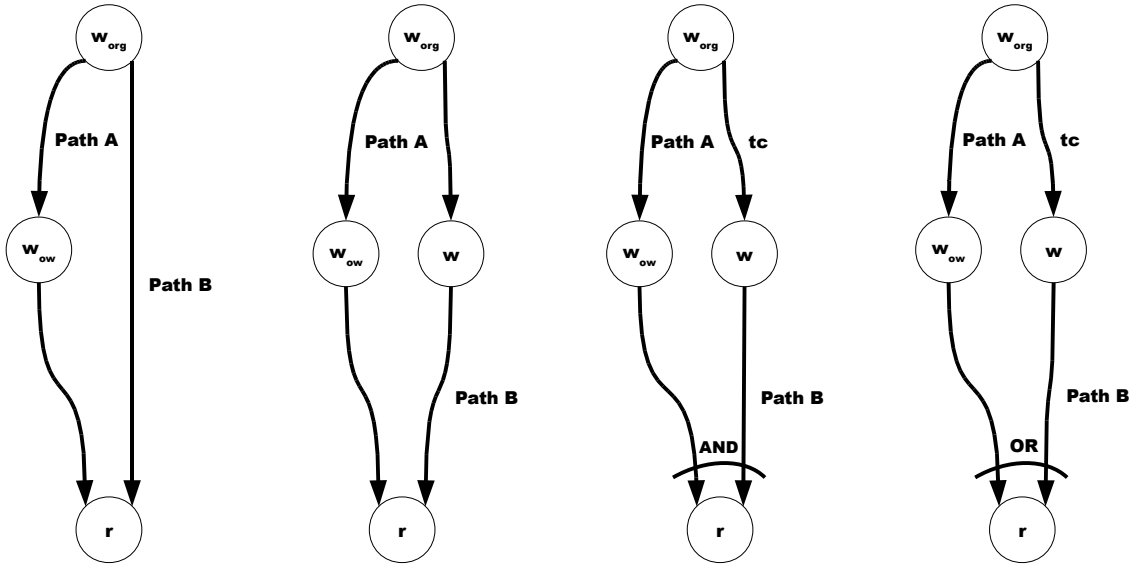
Algorithm 5 Duplication purging (multiple paths)

```
1: procedure PURGEDUPLICATESOR(writes•)
2:     poss• ← poss• \ inv•
3:     dis• ← dis• \ inv•
4:     poss• ← poss• \ dis•
5: end procedure
```

Algorithm 5—PURGEDUPLICATESOR—is one of two algorithms used to purge duplicates after a situation where those can be created. Through the use of multiple outgoing links from the same writer in a `flow` activity, it is possible to have a writer that is in more than one set of writes.

One of the situations is that a writer is both in the poss or dis and the inv set. For this outcome, there are four different situations, depicted in Figure 7.5 on the following page. In all situations, the original writer w_{org} gets overwritten in one path, called path **A**.

1. **Path B has no writers, the original writer stays in** poss**:** In this case, the writer in path **A** has written last. w_{org} is in inv, not in poss.



(a) No writer on path B

(b) Writer on path B, default transition condition

(c) Writer on path B, non-default transition condition, AND-join

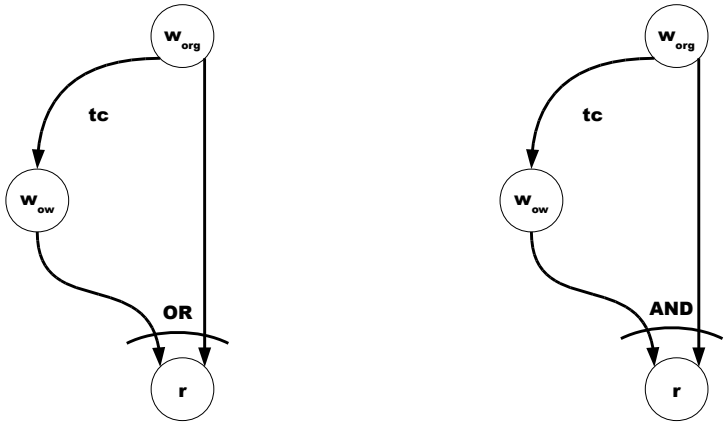
(d) Writer on path B, non-default transition condition, OR-join

Figure 7.5: Emergence of duplications (OR) with inv

2. **Path B has a writer** w **and** $\mathsf{mbd}(w, v_e) = \texttt{false}$**:** Here, w_{org} has been overwritten in both paths, so it is definitely in inv.
3. **Path B has a writer** w **and** $\mathsf{mbd}(w, v_e) = \texttt{true}$**, the joining activity has a non-enabling join condition:** At the joining activity, w_{org} is in dis and in inv; If path B has gone through w at runtime, w_{org} has been completely overwritten. If path B was stopped before w, the outcome is determined by path A, so w_{org} is invalid.
4. **Path B has a writer** w **and** $\mathsf{mbd}(w, v_e) = \texttt{true}$**, the joining activity has an enabling join condition:** Similar to the previous situation; However, w_{org} will not be re-enabled at the current activity, because it is still overwritten from path A. The semantics of re-enablement is "This activity will run and might read a writer prior to the dead path elimination". However, this particular writer w_{org} has been overwritten in another path and thus cannot be re-enabled. Please note that the actual code will re-enable w_{org} but later purge the writer from poss.

Another way of duplication is if a writer is in both poss and dis. For this outcome, there are two different situations, depicted in Figure 7.6 on the next page. In both situations, the original writer w_{org} gets overwritten by another writer with $\mathsf{mbd} = \texttt{true}$ in one path. In that path, w_{org} is pushed into the disabled set. In the other path, it is not changed at all and stays in poss.

1. **The joining activity is re-enabling:** In this case, w_{org} gets re-enabled back into poss. The duplication resolved itself.



(a) OR-join: Re-enable disabled writers

(b) AND-join: Continue dead path

Figure 7.6: Emergence of duplications (OR) with dis

2. **The joining activity is not re-enabling:** Now, the joining activity still lies inside the dead path, so the writer w_{org} cannot stay in poss, because it is still dead. The writer being in dis overrides poss.

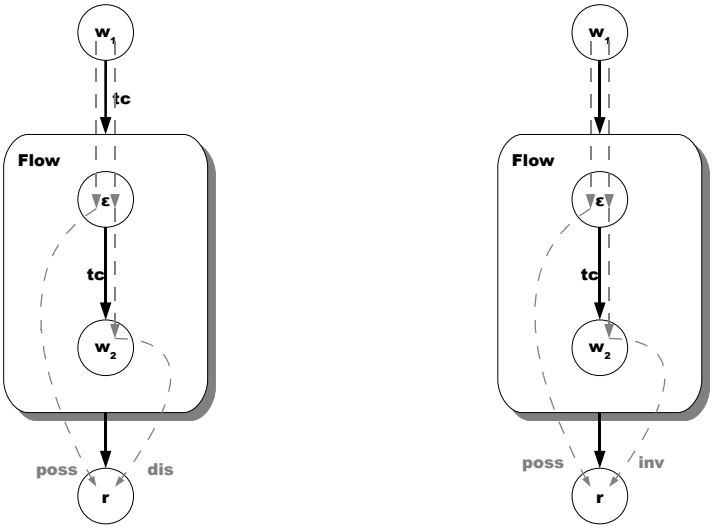
These six situations cover all possibilities of split paths from the same source to the same target. A short explanation for the solution might be: "inv > dis > poss". As can be easily derived, having writers that are in inv already to be taken out of poss and dis (line 2-3) and writers that are in dis to be taken out of poss (line 4) is a way of correctly purging duplicates. This is what happens in Algorithm 5 on page 59. The code is called from MERGEDATA (Algorithm 3 on page 52) only, because this presents the only point where duplicates through joining links can appear.

The procedure is named after the fact that all of the preceding paths will have been executed (if a path is dead, it will still be followed); Duplicates are created because there are multiple paths that all have correct information about writer states. All paths can contribute to the result, the situation mostly resembles a logical *OR*. This is in contrast to the next algorithm.

Algorithm 6 Duplication purging (single path)

```
1: procedure PURGEDUPLICATESXOR(writes•)
2:     inv• ← inv• \ poss•
3:     dis• ← dis• \ poss•
4:     inv• ← inv• \ dis•
5: end procedure
```

Algorithm 6—PURGEDUPLICATESXOR—is the second of the "purge duplicates" algorithms. This piece of code will remove duplicates in a situation where only one of multiple data paths will have been taken; All other paths do not contain valid writer states. This resembles a logical *XOR*. The most ordinary way of this happening are multiple possible paths through an activity, e.g. an `if` or `scope`



(a) Non-default outer transition condition

(b) Default outer transition condition

Figure 7.7: Emergence of duplications (XOR)

activity. If one of the paths is taken at runtime, like the `else` branch of an `if` activity, the others are automatically invalid. On the contrary, if the data paths were actual BPEL links, they would all be equally true.

Since the case of "one path only" (= XOR) is much easier to grasp, we need to look at the different outcomes only instead of whole scenarios, contrary to PURGEDUPLICATESOR (Figure 7.7):

1. **Duplicate writer in** poss **and** dis**:** There is at least one path where the writer is in all cases a possible writer. There is no need to keep the disabled writer, because the paths that might overwrite it are not relevant for static analysis: We already know it is a possible writer. The disabled duplicate can be eliminated.
2. **Duplicate writer in** poss **and** inv**:** As in the case before, there is at least one path where the writer is a possible writer. For the analysis it is irrelevant if there is also a path that overwrites it, because it is possible that path will never be taken, which cannot be statically determined. The writer stays in poss and is removed from the invalid writer set.
3. **Duplicate writer in** dis **and** inv**:** If there is a path that *might* overwrite the original writer and one path that will definitely overwrite it, we keep the disabled writer and purge the invalid writer. The result is much more favorable for the static analysis, because it keeps the writer.
4. **Duplicate writer in all sets:** Again, there is at least one path where the original writer is a possible writer. Static analysis should never report that a writer is invalid or disabled when it *could* be a possible writer, so we purge the duplicates from both dis and inv.

The short version of the result is: "poss $>$ dis $>$ inv". This is what happens in Algorithm 6 on the preceding page.

The only caller of this procedure is overWrite (Algorithm 4 on page 56), which will create duplicate writers when updating writer states of ending container activities.

7.3 Begin and End of Analysis

Algorithm 7 Analysis of a given BPEL process

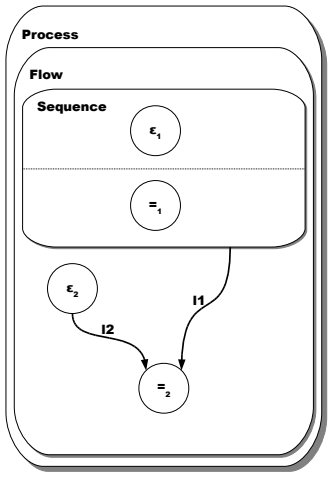
```
 1: procedure ANALYZEPROCESSMODEL(M)
 2:     Determine E_w^{Q_V} in the given BPEL process model M.
 3:     for all v_e ∈ E_w^{Q_V} do
 4:         ∀a ∈ 𝒜 : started(a) ← false
 5:         ∀a ∈ 𝒜 : finished(a) ← false
 6:         ∀ch ∈ ℋ_comp : finished(ch) ← false
 7:         ∀l ∈ ℒ : finished(l) ← false
 8:         HANDLEACTVITY(process, v_e)
 9:     end for
10: end procedure
```

The whole analysis of BPEL processes starts in ANALYZEPROCESSMODEL, as shown in Algorithm 7. In line 2, the set of variable elements that are at any point written in the process is being generated. For each variable element, the whole process model is completely and separately analyzed (line 3-9). To clear any data left by earlier iterations from the process, the started and finished flags of all activities, links and compensation handlers are cleared (line 4-7). After that, analysis of the process activity starts with a call to HANDLEACTIVITY (line 8).

For an efficient implementation, another version of ANALYZEPROCESSMODEL is possible. Iterating over the whole set of variable elements being written to (v_e) and analyzing the whole process for each v_e is inefficient. Instead, the process could be analyzed one single time. The only part of the algorithm actually required to use v_e is "HANDLEBASICACTIVITY", Algorithm 8 on the next page. All other uses only copy v_e so that it is available in HANDLEBASICACTIVITY. Each time HANDLEBASICACTIVITY checks if an activity wrote ($w(a, v_e)$), it could then iterate over all variable elements. In the case that the activity writes to a variable element, the activity is added to poss, like in the presented algorithm, using the matching v_e. This would result in a much more efficient algorithm that only analyzed the process once. The disadvantage in efficiency of HANDLEBASICACTIVITY is negligible.

7.3.1 Analysis Structure

Figure 7.8(a) on the following page contains a BPEL process used to illustrate the execution structure of the algorithm. In Figure 7.8(b) on the next page, the calls of procedures and functions in the course of analyzing this process is documented. Note that only calls concerned with traversing activities and data merging are shown. Also, the call arguments shown are the activities handled only, even if there should be more. The importance of the example lies in the path the analysis takes, not on correct syntax.



(a) BPEL process

```
AnalyzeProcessModel(M)

HandleActivity(process)

HandleActivity(flow)
MergeData(flow)
HandleFlow(flow)

HandleActivity(sequence)
MergeData(sequence)

HandleActivity(empty1)
MergeData(empty1)
HandleBasicActivity(empty1)
HandleSuccessors(empty1)

HandleActivity(assign1)
MergeData(assign1)
HandleBasicActivity(assign1)
HandleSuccessors(assign1)

HandleEndOfSequence(sequence)

HandleSuccessors(sequence)

HandleLink(link1)

HandleActivity(assign2)
// Preconditions (incoming
    links) not satisfied

HandleActivity(empty2)
MergeData(empty2)
HandleBasicActivity(empty2)
HandleSuccessors(empty2)

HandleLink(link2)

HandleActivity(assign2)
MergeData(assign2)
HandleBasicActivity(assign2)
HandleSuccessors(assign2)

HandleEndOfFlow(flow)
```

(b) Series of procedure calls when analyzing the process in Figure 7.8(a)

Figure 7.8: Analysis structure

7.4 Basic Activities

Algorithm 8 Handling a basic activity

1: **procedure** HANDLEBASICACTIVITY(a,v_e)

2: $\mathsf{writes}_\bullet(a, v_e) \leftarrow \begin{cases} \mathsf{overWrite}(\mathsf{writes}_\circ(a, v_e), (\{a\}, \emptyset, \emptyset, \texttt{false})) & w(a, v_e) \\ \mathsf{writes}_\circ(a, v_e) & \text{otherwise} \end{cases}$

3: $\mathsf{finished}(a) \leftarrow \texttt{true}$

4: HANDLESUCCESSORS(a, v_e)

5: **end procedure**

When a basic activity is analyzed, $\mathsf{writes}_\circ$ is updated depending on writes from the activity and written into $\mathsf{writes}_\bullet$. HANDLEBASICACTIVITY is shown in Algorithm 8.

Line 2 shows the main part of HANDLEBASICACTIVITY.

If the activity writes to the variable element currently being analyzed, overWrite (Algorithm 4 on page 56) is called with the writer states previous to the activity execution and a new writes tuple consisting of the basic activity as possible writer only. The result will be a $\mathsf{writes}_\bullet$ tuple where previous possible writers will have been transferred to either $\mathsf{dis}_\bullet$ or $\mathsf{inv}_\bullet$ (depending on the $\mathsf{mbd}_\circ$ state) and a as the new only possible writer.

If the activity does not write to v_e, $\mathsf{writes}_\circ$ is copied to $\mathsf{writes}_\bullet$. It should be noted that this is actually a shortcut for the formal version—calling overWrite with $\mathsf{writes}_\circ$ and $(\emptyset,\emptyset,\emptyset,\texttt{false})$ as arguments. If the possible writers set of the second tuple is empty, overWrite will actually behave in the same way and copy over the original writer states. The shortcut was introduced to improve understandability.

The basic activities writing to a variable element can be determined from [OAS07]. Here is a list of the activities and details regarding the condition $w(a,v_e)$.

- **`assign`:** $w(a,v_e)$ is `true` if v_e is in `to variable` or returned by the expression of `to`.
- **`empty`:** $w(a,v_e)$ is always `false`.
- **`compensate` and `compensateScope`:** $w(a,v_e)$ is always `false`, as compensation activities are separately handled in Algorithm 24 on page 86 "HANDLECOMPENSATE".
- **`exit`:** $w(a,v_e)$ is always `false`.
- **`extensionActivity`:** $w(a,v_e)$ is always `false`, because we do not support extensions.
- **`invoke`:** $w(a,v_e)$ is `true` if v_e is in `outputVariable` or `toVariable`.
- **`receive`:** $w(a,v_e)$ is `true` if v_e is in `variable` or `toVariable`.
- **`reply`:** $w(a,v_e)$ is `true` if v_e is in `faultVariable`.
- **`rethrow`:** $w(a,v_e)$ is always `false`.
- **`throw`:** $w(a,v_e)$ is `true` if v_e is in `faultVariable`.
- **`validate`:** $w(a,v_e)$ is always `false`, because `validate` does not write to any BPEL variables.
- **`wait`:** $w(a,v_e)$ is always `false`, because `wait` does not write to any BPEL variables.

As discussed in Section 4.2 on page 28, a writer always writes to the specified variable *and* all sub-elements of the variable.

Since the activity is now completely analyzed, finished can be set to `true` (line 3). Analysis continues with outgoing links, sequential successors and containing activities in HANDLESUCCESSORS (line 4).

7.5 `flow` Activity

HANDLEFLOW and HANDLEENDOFFLOW analyze the beginning and end of `flow` activities.

Algorithm 9 Handling a `flow` activity

```
1: procedure HANDLEFLOW(f,v_e)
2:     roots ← {a | a ∈ children(f) ∧ ¬∃l : (a',l,a) ∈ LR, a' ∈ descendants(f)}
3:     for all a_r ∈ roots do
4:         HANDLEACTIVITY(a_r,v_e)
5:     end for
6: end procedure
```

HANDLEFLOW in Algorithm 9 on the preceding page is used to start analysis of the root activities of the `flow`. From this procedure, the rest of the `flow` (deeply contained as well as directly contained activities) will be reached. Discovering root activities of a `flow` is straightforward: Find all contained activities of the `flow` that have no incoming links from any descendant of the `flow` (line 2). In lines 3 to 5 a loop iterates over these roots that starts analysis for each one (line 4) using HANDLEACTIVITY (see Algorithm 1 on page 49). Note that activities contained in `flow`s are specially handled because of their connecting links. See Section 7.1 on page 48 and Section 7.2 on page 50 for that.

Algorithm 10 Handling the end of a `flow` activity

```
1: procedure HANDLEENDOFFLOW(f, v_e)
2:     leaves ← {a | a ∈ children(f) ∧ ¬∃l : (a, l, a') ∈ LR, a' ∈ descendants(f)}
3:     if ∀a_l ∈ leaves : finished(a_l) then
4:         writes•(f, v_e) ← ⨆_{a_l ∈ leaves} writes•(a_l, v_e)
5:         writes•(f, v_e) ← overWrite(writes∘(f, v_e), writes•(f, v_e))
6:         HANDLESUCCESSORS(f, v_e)
7:     end if
8: end procedure
```

HANDLEENDOFFLOW is called after any leaf activity of a `flow` ended and called for successor handling (see HANDLESUCCESSORS, Algorithm 2 on page 51). Line 2 determines all leaf activities of the `flow`: All activities that are direct children and have no outgoing links to descendants of the `flow`. The $\text{writes}_\bullet$ of the `flow` are the merged writers of all leaves, because the moment the `flow` ends, those leaf activities will be the last to have been executed inside. Of course, to correctly end the `flow`, all leaves have to be checked to make sure they have already been handled (line 3). If they are, the algorithm can go on with the analysis. If they are not, the algorithm stops and will be called later from any other leaf activity.

The merging of writer states from all leaves happens in line 4. Note that this first merging is the collection of data inside of the `flow` only. So far, there is no update to the data that came from the outside, which happens during overWrite (Algorithm 4 on page 56). What is now contained in the $\text{writes}_\bullet$ tuple is the final state of writers when the `flow` ends, not regarding any writers previous to the `flow`.

In line 5 the $\text{writes}_\circ$ data of the `flow` is updated with the new writers from the $\text{writes}_\bullet$ of the inside of the `flow`. This will merge in the formerly removed disabled writers, put previous possible writers into the disabled or invalid set and add anything that has happened on the inside. The result is a $\text{writes}_\bullet$ that has possible, disabled and invalid writers after the `flow` ended, also containing activities previous to the `flow`.

The call to HANDLESUCCESSORS in line 6 ensures that the analysis of successors such as sequential activities, outgoing links or other “end of container” handlers will follow.

Algorithm 11 Handling a link

```
 1: procedure HANDLELINK(l, v_e)
 2:     s,t : (s,l,t) ∈ LR
 3:     p_s, p_t : (p_s,s) ∈ HR, (p_t,t) ∈ HR
 4:     p_st ← sharedparent(s,t)
 5:     writes_∘(l,v_e) ← writes_•(s,v_e)
 6:     writes_•(l,v_e) ← writes_∘(l,v_e)
 7:
 8:     // Traverse hierarchy
 9:     if p_st ≠ p_s then
10:         p ← p_s
11:         while p ≠ p_st do
12:             writes_•(l,v_e) ← overWriteNoEndDeadPath(writes_∘(p,v_e), writes_•(l,v_e))
13:             p ← p_τ, (p_τ,p) ∈ HR
14:         end while
15:     end if
16:     mbd_•(l,v_e) ← mbd_•(l,v_e) ∨ ¬alwaystrue(tc(l))
17:     finished(l) ← true
18:     HANDLEACTIVITY(t,v_e)
19: end procedure
```

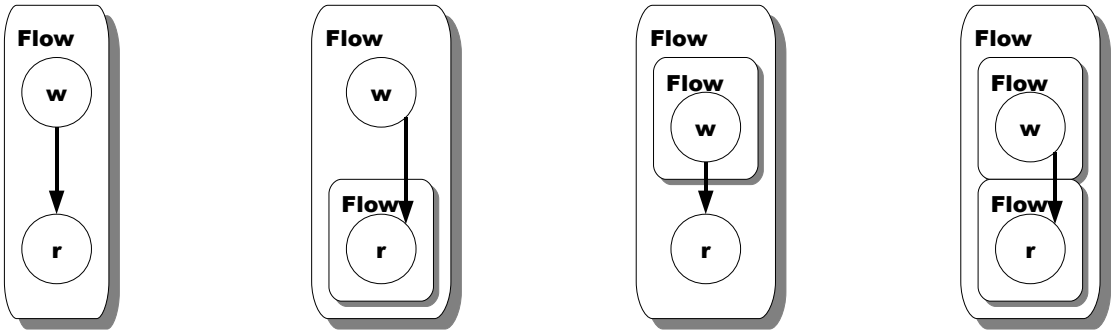
7.6 BPEL Links

Algorithm 11 displays the procedure HANDLELINK. This procedure is called from HANDLESUCCESSORS (Algorithm 2 on page 51) when any activity (basic or container) has successfully ended its analysis.

The procedure takes a link l and a variable element v_e, both of which are currently being analyzed, as arguments. In [KKL07c], HANDLELINK copied over the writes tuple from the source activity and changed mbd according to the transition condition of the link. My version of this algorithm still does that, but adds logic for links crossing the boundaries of container activities, called "cross-boundary links".

The problem with cross-boundary links is the scoping that the analysis algorithm does concerning container activities. To properly analyze dead paths, the analysis drops disabled writers when entering analysis of a container activity and merges them again after the container ended; See Algorithm 3 on page 52 "MERGEDATA" (Section 7.2 on page 50) and Algorithm 12 on page 69 "overWriteNoEndDeadPath" for more information.

Now when a link is boundary-crossing, it will have a part of the actual writer states only, because its source lies within a container. To merge in the writers prior to the container when a link exits said container, the function overWriteNoEndDeadPath can be used. The first argument are the writers previous to the container currently being exited. The second argument are the writers that were active before the link, missing disabled writers from outside the container. This way, if a link crosses multiple



(a) No cross-boundary link

(b) Cross-boundary incoming link

(c) Cross-boundary outgoing link

(d) Cross-boundary incoming and outgoing link

Figure 7.9: Variations of cross-boundary links

boundaries, we need to call overWriteNoEndDeadPath with each successive parent container and whatever writers the link itself "carries".

In the first five lines from 2 to 6, the analysis is set up. Line 2 determines the source and target activities s and t of the link through the LR relation. In line 3 the parents of those activities are taken from HR; p_s is the parent of the source activity s, p_t is the parent of the target activity t. After that, we need to find the innermost shared parent p_{st} of both activities in line 4. The setup part finishes with copying the source activity's writers to $\mathsf{writes}_\circ$ and $\mathsf{writes}_\bullet$ of the link in lines 5 and 6.

The shared parent is needed to see how many containers we have to leave from the source to the target. There are four situations (see Figure 7.9):

1. **s and t are direct children of the same container:** p_{st} and p_s are the same, no cross-boundary handling. l is not boundary-crossing.
2. **t is a descendant of s's parent:** p_{st} and p_s are the same. l is boundary-crossing, but not outgoing. Incoming cross-boundary links do not need special handling, so we can handle this case like the first one.
3. **s is a descendant of t's parent:** p_{st} and p_t are the same. This is an outgoing cross-boundary link and needs to be specially handled, as described below.
4. **Neither s nor t are descendants of each other's direct parents:** p_{st} is not the same as p_s or p_t. Rather, p_{st} is at most the direct child of the `process` element.

All four situations can be collapsed into two steps: One, check whether the link is actually a cross-boundary outgoing link by checking if $p_{st} \neq p_s$. Two, run overWriteNoEndDeadPath for each container between the source activity and the target activity (excluding p_t), the control variable being p. This happens in line 9 and line 10-14. The call to overWriteNoEndDeadPath is in line 12 and will always update $\mathsf{writes}_\bullet(l, v_e)$.

Lines 16 and 17 contain additional and necessary handling. $\mathsf{mbd}_\bullet$ is updated to reflect the fact that the link's transition condition will start a dead path, if the condition might be `false`. finished is set to `true`.

In the last line 18, HANDLEACTIVITY is called to start analysis of the target activity.

Algorithm 12 Data overwriting without ending dead paths

```
 1: function overWriteNoEndDeadPath(writes_○, writes_τ)
 2:     if poss_τ = ∅ then
 3:         writes_• ← writes_○
 4:     else if poss_τ ≠ ∅ ∧ mbd_○ then
 5:         writes_• ← (poss_τ, dis_○ ∪ poss_○ ∪ dis_τ, inv_○ ∪ inv_τ, poss_○ ∩ poss_τ ≠ ∅)
 6:     else if poss_τ ≠ ∅ ∧ ¬mbd_○ then
 7:         writes_• ← (poss_τ, dis_○ ∪ dis_τ, inv_○ ∪ inv_τ ∪ poss_○, false)
 8:     end if
 9:     PURGEDUPLICATESXOR(writes_•)
10:     return writes_•
11: end function
```

Algorithm 12 shows function overWriteNoEndDeadPath. This piece of code is used to update the writer states of the link when it crosses container boundaries. To get a full explanation of the code, refer to Algorithm 4 on page 56 "overWrite".

overWriteNoEndDeadPath is a variant of overWrite. The difference is in the way dis_τ is handled. Disabled writers from inside a container activity need to be re-enabled when the container activity ends, as the dead path does not continue after a container ended. This re-enabling cannot be applied when analyzing cross-boundary links, as the dead path continues through the link. The link will always stay inside the dead path until a re-enabling join condition appears (see Algorithm 3 on page 52). Disabled writers will stay disabled.

The same reasoning applies to determining $\mathsf{mbd}_\bullet$. If there is a dead path that disabled a writer placed outside of the container activity (that is, dis_τ and $\mathsf{poss}_\circ$ overlap), this dead path is still active. Thus, only overlapping of $\mathsf{poss}_\circ$ and poss_τ can determine $\mathsf{mbd}_\bullet$.

7.7 `sequence` Activity

Sequence activities are mostly handled by unspecific code. To start the analysis of the first activity in a sequence, HANDLEACTIVITY (Algorithm 1 on page 49) lines 29-31 determine the first activity and apply the analysis to it. MERGEDATA—Algorithm 3 on page 52—is also specially written to cope with sequences (see Section 7.2 on page 50).

Starting and continuing a sequence is already covered. What is not covered is the return to the previous container once the last activity of a sequence has been analyzed—ending the sequence. How this is accomplished is shown in Algorithm 13 on the following page.

Algorithm 13 Handling the end of a sequence activity

```
1: procedure HANDLEENDOFSEQUENCE(s, v_e)
2:     a : (s,a) ∈ HR, (a,a') ∉ SR
3:     writes•(s, v_e) ← writes•(a, v_e)
4:     writes•(s, v_e) ← overWrite(writes∘(s, v_e), writes•(s, v_e))
5:     HANDLESUCCESSORS(s, v_e)
6: end procedure
```

Since HANDLEENDOFSEQUENCE is called only if all preconditions are true (i. e. the last activity has been successfully handled), there is no need to check them again. The first argument to this procedure is the sequence activity itself, so we have to determine the finishing activity. This happens in line 2: Find a direct child of the sequence that has no successor in the sequence relation SR.

The next command, in line 3 will copy the $\mathsf{poss}_\bullet$ data of the last sequence activity to the end of the sequence. Since a contained dead path ends here (as in each activity of a sequence), disabled writers need to be re-enabled, but that is handled in function overWrite (Algorithm 4 on page 56).

Line 4 is the general "leave container and merge writers prior to container" already explained in earlier sections. The result is a $\mathsf{writes}_\bullet$ tuple that presents the state of writers after the sequence has completed.

After this, the successors—such as outgoing links, sequential successors or ending containers—are analyzed through the use of HANDLESUCCESSORS in line 5.

7.8 Loop Activities

Loop activities pose a special challenge to static analysis: There is no way to determine statically how often the loop will run. In the presented algorithm, this challenge is overcome by running analysis repeatedly. As the process is only abstractly interpreted, the only relevant information contained in the loops is the order of writing activities. When running analysis repeatedly, a fix point is tried to be reached, when the writer state does not change in continuing runs. This analysis loop will definitely end, as the set of possible writers is a finite set. When all writing activities of the process are possible writers, the writer state will not change, and the analysis loop will end.

In all observed cases, loop analysis will end after two loops.

There are three kinds of looping activities that can be handled by the procedures in this section: `while`, `repeatUntil` and sequential `forEach` activities. Loop analysis is the same in all cases.

Analysis is started by calling HANDLEACTIVITY (Algorithm 1 on page 49) on the main activity. After finishing this, HANDLEENDOFLOOP is called by HANDLESUCCESSORS (Algorithm 2 on page 51). There, it is decided whether loop analysis has to go on or if it can stop at this point.

HANDLEENDOFLOOP is shown in Algorithm 14 on the facing page.

Algorithm 14 Handling looping or ending `while`/`repeatUntil`/`forEach` loops

procedure HANDLEENDOFLOOP(l, v_e)
  $c : (l, c) \in \mathsf{HR}$
  if $\mathsf{writes}_\circ(l, v_e) = \mathsf{writes}_\bullet(c, v_e)$ **then**
    $\mathsf{writes}_\bullet(l, v_e) \leftarrow \mathsf{overWrite}(\mathsf{writes}_\circ(l, v_e), \mathsf{writes}_\bullet(c, v_e))$
    HANDLESUCCESSOR(l, v_e)
  else
    $\mathsf{writes}_\circ(l, v_e) \leftarrow \mathsf{overWrite}(\mathsf{writes}_\circ(l, v_e), \mathsf{writes}_\bullet(c, v_e))$
    HANDLEACTIVITY(c, v_e)
  end if
end procedure

HANDLEENDOFLOOP gets called with the loop l and a variable element v_e. The first thing that has to be done is to determine the contained activity n that will be looped (line 2).

The main part of the procedure consists of an if-else-branching. Either, loop analysis can be stopped here, because $\mathsf{writes}_\circ(l, v_e) = \mathsf{writes}_\bullet(n, v_e)$ (line 3)—in other words: The writer state has not changed through the last analysis step. In that case, $\mathsf{writes}_\bullet$ of the loop can be set by updating $\mathsf{writes}_\circ$ of the loop with $\mathsf{writes}_\bullet$ of the contained activity (line 4). Analysis of any loop successors is started in line 5.

If the writer state did change in the last analysis step (line 6), said state is copied into $\mathsf{writes}_\circ$ of the loop. This way, $\mathsf{writes}_\circ$ itself is updated using the changes from the contained activity analysis (line 7). After that, HANDLEACTIVITY is called using the contained activity n and variable element v_e in line 8.

7.9 `pick` Activity

`pick` activities are branching activities. At runtime, one of the branches is taken. At analysis time, it cannot be statically determined which branch is executed. As a result, all branches are analyzed separately. The writers valid before each branch are the writers valid before the `pick` activity. The analysis result of each branch is merged into the analysis result of the `pick` activity.

Listing 7.1 Outline of a `pick` activity

```
1  <pick>
2    <onMessage>
3      activity
4    </onMessage>
5    ...
6    <onAlarm>
7      activity
8    </onAlarm>
9    ...
10 </pick>
```

Listing 7.1 on the previous page shows an outline of a `pick` activity for reference.

Algorithm 15 Handling `pick` activities

```
1:  procedure HANDLEPICK(a_pick, v_e)
2:      for all branch : (a_pick, branch) ∈ HR do
3:          a_branch : (branch, a_branch) ∈ HR
4:          writes∘(branch, v_e) ← (poss∘(a_pick, v_e), ∅, inv∘(a_pick, v_e), false)
5:          if w(branch, v_e) then
6:              writes∘(branch, v_e) ← overWrite(writes∘(branch, v_e), ({branch}, ∅, ∅, false))
7:          end if
8:          HANDLEACTIVITY(a_branch, v_e)
9:      end for
10: end procedure
```

Algorithm 15 "HANDLEPICK" presents the start of the analysis of a `pick` activity. It is called from HANDLEACTIVITY (Algorithm 1 on page 49).

Line 2 starts an iteration over all branches of the `pick` activity. These branches are either `onMessage` or `onAlarm` branches. Branches are all elements contained in a `pick` activity, according to HR.

Line 3 retrieves the activity contained in each branch. The contained activity is the only activity in each branch, according to HR.

Line 4 copies the writers from before the `pick` activity into the branches. Disabled writers are not copied, as they cannot be re-enabled inside the branch. Also, the $\mathsf{mbd}_\circ$ state is not copied. These two facts stem from the scoping applied because of DPE, as described in Section 4.5 on page 31. In other cases, this would be achieved by HANDLEACTIVITY (Algorithm 1) calling MERGEDATA (Algorithm 3 on page 52). In this case, because the `onMessage` branches may write, the scoping has to be applied earlier.

The data is copied to the branch only, not to the contained activity. The activity analysis retrieves the data itself, by using MERGEDATA (Algorithm 3).

`onMessage` branches behave like `receive` activities, which means they can be writers, although they are not basic activities. Line 5 checks if the branch is a writer. A branch is a writer ($w(a, v_e)(branch, v_e) = \texttt{true}$) if v_e is in the `variable` attribute or in the `toVariable` attribute of the `fromParts` element of the `pick` activity.

If that is the case, line 6 adds the the branch as a writer. It calls overWrite (Algorithm 4 on page 56) to update $\mathsf{writes}_\circ$ of the branch.

Line 8 calls HANDLEACTIVITY (Algorithm 1) to analyze the activity contained in the branch.

This concludes Algorithm 15 "HANDLEPICK".

Algorithm 16 on the next page "HANDLENDOFPICK" is the second algorithm necessary to analyze `pick` activities. It is called from HANDLESUCCESSORS (Algorithm 2 on page 51). It is called each time an activity contained inside a `pick` activity has finished analysis.

Algorithm 16 Handling ending pick activities

1: **procedure** HANDLEENDOFPICK(a_{pick}, v_e)
2: **if** $\forall a_{branch} : (a_{pick}, branch) \in \mathsf{HR}, (branch, a_{branch}) \in \mathsf{HR}, \mathsf{finished}(a_{branch})$ **then**
3: $\mathsf{writes}_\bullet(a_{pick}, v_e) \leftarrow \bigsqcup_{a:(branch,a_{branch})\in\mathsf{HR},(a_{pick},branch)\in\mathsf{HR}} \mathsf{writes}_\bullet(a, v_e)$
4: $\mathsf{writes}_\bullet(a_{pick}, v_e) \leftarrow \mathsf{overWrite}(\mathsf{writes}_\circ(a_{pick}, v_e), \mathsf{writes}_\bullet(a_{pick}, v_e))$
5: **end if**
6: **end procedure**

Line 2 checks for each activity contained in the `pick` activity if it has finished analysis. This is achieved by iterating over all branches and retrieving all contained activities, according to HR. For each contained activity, the finished flag is checked, as described in Section 4.3 on page 29.

If not all contained activities have finished analysis, line 2 yields `false`. No further processing is currently done for the `pick` activity. All contained activities need to be finished with analysis before the `pick` activity can be allowed to finish analysis. When another contained activity of the `pick` activity finishes analysis, HANDLEENDOFPICK (Algorithm 16) is called again.

If all contained activities have finished analysis, processing of the `pick` activity can continue. Line 3 retrieves and merges the $\mathsf{writes}_\bullet$ data of all contained activities of the `pick` activity. The result is written to $\mathsf{writes}_\bullet$ of the `pick` activity.

Line 4 updates the writer state valid before the analysis of the `pick` activity with the analysis data of contained activities. This is achieved using overWrite (Algorithm 4 on page 56). This also re-enables contained disabled writers, sets $\mathsf{mbd}_\bullet$ and purges duplicate writers.

This concludes Algorithm 16 "HANDLEENDOFPICK".

7.10 `if` Activity

`if` activities are branching activities. They are very similar in analysis to `pick` activities. At runtime, one of the branches is executed. At analysis time, it cannot be statically determined which branch is taken. As a result, each branch is analyzed separately. The writers valid before each branch are the writers valid before the `if` activity. The analysis result of each branch is merged into the analysis result of the `if` activity.

Listing 7.2 on the next page shows an outline of an `if` activity for reference.

Algorithm 17 on the following page "HANDLEIF" presents the start of analysis for an `if` activity. It is called from HANDLEACTIVITY (Algorithm 1 on page 49).

Line 2 determines the activity that is executed if the `if` condition yields `true`. This activity is called the "main activity". This activity is the only activity directly contained inside the `if` activity, according to HR and $\mathscr{A}$.

Listing 7.2 Outline of an `if` activity

```
1  <if>
2    activity
3    <elseif>
4      activity
5    </elseif>
6    ...
7    <else>
8      activity
9    </else>
10 </if>
```

Algorithm 17 Handling `if` activities

1: **procedure** HANDLEIF(a_{if}, v_e)
2: $a_{main} : (a_{if}, a_{main}) \in \mathsf{HR}, a_{main} \in \mathscr{A}$
3: $else : (a_{if}, else) \in \mathsf{HR}, else \notin \mathscr{A}$
4: HANDLEACTIVITY(a_{main}, v_e)
5: **for all** $e : e \in else$ **do**
6: $\mathsf{writes}_\circ(e, v_e) \leftarrow \mathsf{writes}_\circ(a_{if}, v_e)$
7: HANDLEACTIVITY$(a_{else}, v_e), (e, a_{else}) \in \mathsf{HR}$
8: **end for**
9: **end procedure**

Line 3 determines all `elseif` and `else` branches. These are all elements contained in the `if` activity that are not activities, according to HR and $\mathscr{A}$.

Line 4 starts analysis of the main activity by calling HANDLEACTIVITY (Algorithm 1).

Line 5 iterates over all `elseif` and `else` branches.

Line 6 copies the writer state valid before the `if` activity started analysis. The writer state is copied to the branches only. MERGEDATA (Algorithm 3 on page 52) ensures that the contained activities receive the data.

Line 7 starts the activity analysis for each branch activity by calling HANDLEACTIVITY (Algorithm 1).

This concludes Algorithm 17 "HANDLEIF".

Algorithm 18 on the next page "HANDLEENDOFIF" is called by HANDLESUCCESSORS (Algorithm 2 on page 51). It is called when one of the activities contained in an `if` activity have finished analysis.

Line 2 determines the main activity. This is the only activity contained in the `if` activity according to HR that is also in $\mathscr{A}$.

Line 3 determines the set of `elseif` and `else` branches of the `if` activity.

Line 4 checks if all contained activities have finished analysis. This is the case if the finished flag of all activities is set to `true`.

Algorithm 18 Handling ending `if` activities

1: **procedure** HANDLEENDOFIF(a_{if}, v_e)
2: $a_{main} : (a_{if}, a_{main}) \in \mathsf{HR}, a_{main} \in \mathscr{A}$
3: $else : (a_{if}, else) \in \mathsf{HR}, else \notin \mathscr{A}$
4: **if** $\mathsf{finished}(a_{true}) \wedge \forall a_{else}, (else, a_{else}) \in \mathsf{HR} : \mathsf{finished}(a_{else})$ **then**
5: $\mathsf{writes}_\bullet(a_{if}, v_e) \leftarrow \mathsf{writes}_\bullet(a_{main}, v_e) \sqcup \bigsqcup_{a \in a_{else}} \mathsf{writes}_\bullet(a, v_e)$
6: $\mathsf{writes}_\bullet(a_{if}, v_e) \leftarrow \mathsf{overWrite}(\mathsf{writes}_\circ(a_{if}, v_e), \mathsf{writes}_\bullet(a_{if}, v_e))$
7: **end if**
8: **end procedure**

If not all contained activities have finished analysis, processing stops here. HANDLEENDOFIF (Algorithm 18) will be called again after analysis of another contained activity has been finished.

If all contained activities have finished analysis, processing can continue.

Line 5 merges the analysis results of all contained activities. This is accomplished by copying and merging the $\mathsf{writes}_\bullet$ tuples of each contained activity.

Line 6 updates the writer state. This is achieved by overwriting the writer state that was valid before the `if` activity with the merged writer state of the contained activities. overWrite (Algorithm 4 on page 56) is used for the update. This also re-enables disabled writers contained inside the `if` activity, determine $\mathsf{mbd}_\bullet$ and purge duplications in $\mathsf{writes}_\bullet$.

This concludes Algorithm 18 "HANDLEENDOFIF".

7.11 `scope` Activity

`scope` activities are container activities. They contain exactly one main activity. They contain fault, termination, event and compensation handlers.

If the main activity faults, the fault handler is executed. If the `scope` is induced to terminate, all activities are stopped and the termination handler is called. The compensation handler is called by a `compensate` or `compensateScope` command outside of the `scope`. Event handlers can execute at any time of the `scope` running in normal mode (as opposed to a faulted `scope`).

At runtime, the fault, termination and event handlers can be executed at any time. This implies that analysis of all activities inside the `scope` activity needs to be finished before analysis of the handlers begins.

There is no "HANDLESCOPE" activity. The contained activity of a `scope` activity—the "main activity"—is analyzed directly from HANDLEACTIVITY (Algorithm 1 on page 49).

Listing 7.3 on the next page presents the outline of a `scope` activity for reference.

Listing 7.3 Outline of a `scope` activity

```
1  <scope>
2    activity
3
4    <eventHandlers>
5      <onEvent>
6        activity
7      </onEvent>
8      <onAlarm>
9        activity
10     </onAlarm>
11     ...
12   </eventHandlers>
13
14   <faultHandlers>
15     <catch>
16       activity
17     </catch>
18     ...
19     <catchAll>
20       activity
21     </catchAll>
22   </faultHandlers>
23
24   <terminationHandler>
25     activity
26   </terminationHandler>
27
28   <compensationHandler>
29     activity
30   </compensationHandler>
31 </scope>
```

Algorithm 19 Analysis of `scope` handlers

```
1: procedure HANDLESCOPEHANDLERS(a_scope, v_e)
2:     HANDLEFAULTHANDLERS(a_scope, v_e)
3:     HANDLEEVENTHANDLERS(a_scope, v_e)
4:     HANDLETERMINATIONHANDLER(a_scope, v_e)
5: end procedure
```

Algorithm 19 "HANDLESCOPEHANDLERS" presents the starter algorithm for analyzing a `scope`'s handlers. It is called from HANDLESUCCESSORS (Algorithm 2 on page 51). It is called after the main activity contained in a `scope` has finished analysis.

Line 2 calls HANDLEFAULTHANDLERS (Algorithm 20 on the facing page) to analyze the fault handlers of the `scope`.

Line 3 calls HANDLEEVENTHANDLERS (Algorithm 21 on page 80) to analyze the event handlers of the `scope`.

Line 4 calls HANDLETERMINATIONHANDLER (Algorithm 22 on page 81) to analyze the termination handler of the scope.

This concludes Algorithm 19.

7.11.1 Fault Handlers

Algorithm 20 Analysis of fault handlers

1: **procedure** HANDLEFAULTHANDLERS($a_{\texttt{scope}}$, v_e)
2: $a_{main} : (a_{\texttt{scope}}, a_{main}) \in \mathsf{HR}, a_{main} \in \mathscr{A}$
3: **for all** $x : (x \in \mathsf{descendants}(a_{main}) \vee x = a_{main}) \wedge \mathsf{localtoscope}(x, a_{\texttt{scope}}) \wedge x \notin \mathscr{A}_{scope}$ **do**
4: $\mathsf{writes}_\tau \leftarrow \mathsf{writes}_\tau \sqcup \mathsf{writes}_\circ(x, v_e) \sqcup \mathsf{writes}_\bullet(x, v_e)$
5: **end for**
6: **for all** $x : (x \in \mathsf{descendants}(a_{main}) \vee x = a_{main}) \wedge \mathsf{localtoscope}(x, a_{\texttt{scope}}) \wedge x \in \mathscr{A}_{scope}$ **do**
7: $a_{term} : (h_{term}, a_{term}) \in \mathsf{HR}, (x, h_{term}) \in \mathsf{HR}, h_{term} \in \mathscr{H}_{term}$
8: $\mathsf{writes}_\tau \leftarrow \mathsf{writes}_\tau \sqcup \mathsf{writes}_\bullet(a_{term}, v_e)$
9: **end for**
10: $\mathsf{writes}_\tau \leftarrow \left(\mathsf{poss}_\tau, \emptyset, \mathsf{inv}_\tau \cup \mathsf{dis}_\tau, \texttt{false}\right)$
11: PURGEDUPLICATESXOR(writes_τ)
12: **for all** $h_{fault} : (container_{fh}, h_{fault}) \in \mathsf{HR}, (a_{\texttt{scope}}, container_{fh}) \in \mathsf{HR}, h_{fault} \in \mathscr{H}_{fault}$ **do**
13: $a_{fault} : (h_{fault}, a_{fault}) \in \mathsf{HR}$
14: $\mathsf{writes}_\circ(h_{fault}, v_e) \leftarrow \mathsf{writes}_\tau$
15: HANDLEACTIVITY(a_{fault}, v_e)
16: **end for**
17: **end procedure**

Algorithm 20 "HANDLEFAULTHANDLERS" presents the start of fault handler analysis. It is called by HANDLESUCCESSORS (Algorithm 2 on page 51) via HANDLESCOPEHANDLERS (Algorithm 19 on the preceding page) after the main activity contained in a scope has been analyzed.

Line 2 retrieves the main activity contained in the scope. This is needed as the data for the fault handlers come from the activities contained in the main activity, or from itself, if it is a basic activity.

Line 3 iterates over all non-scope activities contained in the main activity (or itself) that are local to the scope. $\mathsf{descendants}(a_{main})$ returns a set of all activities contained inside the main activity.

The function localtoscope takes two arguments, both activities. It yields true if the first activity is local to the scope activity given as second argument. It yields false if the first activity is not local to the scope activity given as second argument. "Being local to a scope activity" implies that there is no other scope activity containing the first activity, that is also contained inside the given scope activity. For fault handlers, this basically means that the given activity will call the given scope's fault handlers when a fault occurs.

The activities to iterate over cannot be scope activities because these require special handling.

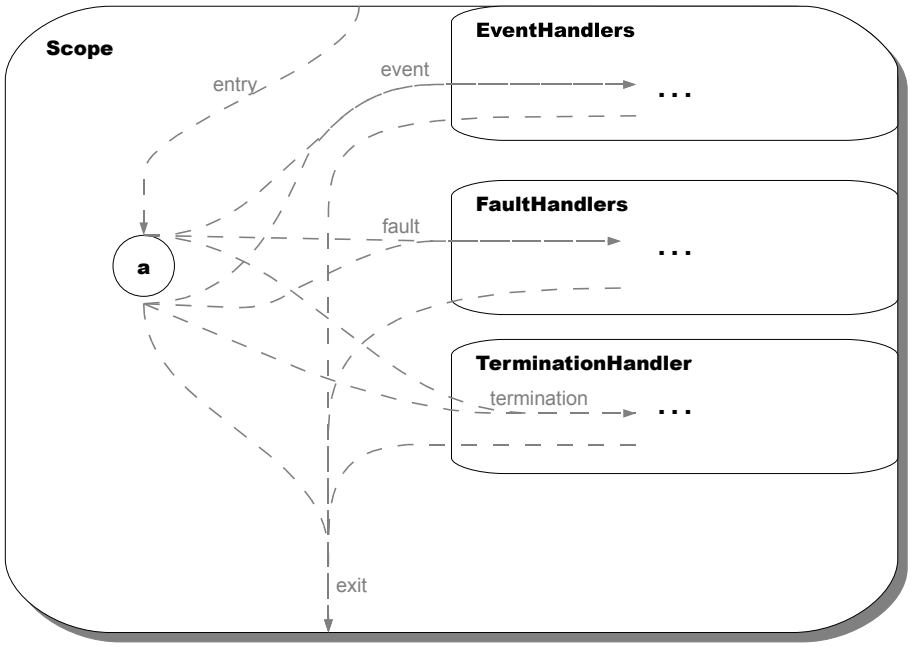


Figure 7.10: Accumulation of data flow edges in a basic `scope`

When a fault occurs, it is not statically decidable where and when it occurred. Additionally, activities parallel to the faulting activity are terminated. These activities might have written to the variable element before they terminate, so these writers need to be taken into account as well. Due to the undecidability of fault occurrence and termination, a data flow edge is created from each activity local to the `scope` activity. Activities of enclosed `scopes` are handled differently.

As depicted in Figure 7.10, this over-approximation can create a huge amount of data flow edges. This is unfortunate, but currently there is no heuristic to improve this part of the algorithm.

Line 4 acquires the writer state before and after the activities local to the `scope` activity. These writer states are contained in $\mathsf{writes}_{\circ}$ and $\mathsf{writes}_{\bullet}$. They are merged into writes_{τ}, a temporary writer state tuple. Writer state before and after each activity is needed, again, because of the static undecidability of faults and termination: Some activities write before faulting or terminating, some do not.

Line 6 starts the next loop, which iterates over all `scope` activities local to the analyzed `scope` activity.

Line 7 retrieves the termination activity of each `scope` activity. This is achieved by traversing the HR relation through the termination handler h_{term}.

Line 8 merges the writer state valid after the termination handler executed into the temporary variable writes_{τ}. When a `scope` activity faults, all activities contained in the main activity (and itself) terminate. For contained `scope` activities, after termination of the contained activities, the termination handler is executed. This implies that for contained `scope` activities, we can seek the writer state of the

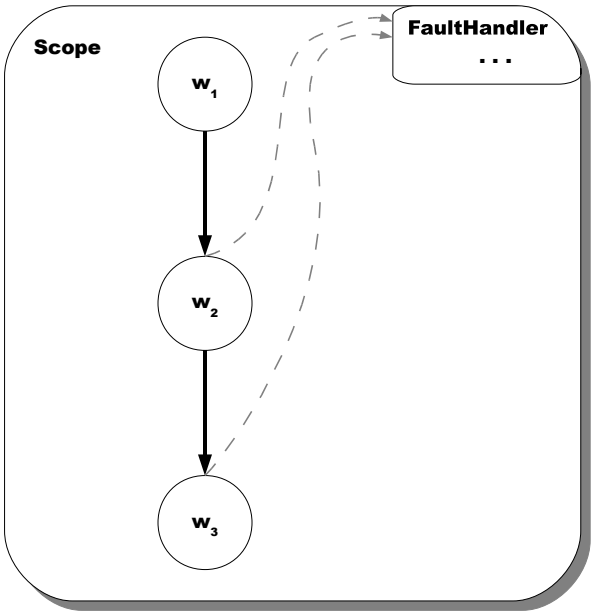


Figure 7.11: Multiple ways to start the Fault Handler might introduce duplicate writers; In this case, w_1 is both in poss as well as in inv.

termination handler to examine the effects of forced termination. The termination handlers of the contained scope is in all cases already finished in analysis, as the fault handler of the outer scope activity is only analyzed after the main activity (and contained activities) are completely analyzed.

Line 10 places all disabled writers into the invalid writers set. When stopping data flow at any arbitrary activity inside the scope activity, sometimes the last writer active was in a dead path. The disabled writer that was also copied is in that case the writer active before the dead path. Analysis of the fault handler is only interested in the last writer active, but can never re-enable the writer prior to a dead path. Thus, the disabled writers can be moved to the invalid writers, so they will not be re-enabled at any point. By the same statement, mbd_τ is set to `false`. If writer state was retrieved from any place that may be dead, this is no concern for the fault handler. It will be executed nonetheless.

Line 11 calls PURGEDUPLICATESXOR (Algorithm 6 on page 61). This eliminates all duplication in the writes_τ tuple. Writers that are already in poss_τ override writers in inv_τ. dis_τ has already been cleared. The reasoning is simple: If the fault handler receives writer data from two places, with a writer being in poss in one place and in inv in another, then there is at least one possibility that the writer is active at the execution of the fault handler. This is depicted in Figure 7.11.

Line 12 iterates over all fault handlers of the scope activity. This is accomplished by finding all elements in $\mathscr{H}_{fault}$ that are contained in the "fault handler container" ($container_{fh}$) of the scope activity.

Line 13 retrieves the activity contained in the fault handler.

Line 14 copies the writer state determined earlier to the fault handler. MERGEDATA (Algorithm 3 on page 52) will be concerned with acquiring the data for the fault activity before analysis starts.

Line 15 starts activity analysis on the fault activity by calling HANDLEACTIVITY (Algorithm 1 on page 49).

This concludes Algorithm 20.

7.11.2 Event Handlers

Algorithm 21 Analysis of event handlers

1: **procedure** HANDLEEVENTHANDLERS($a_{\texttt{scope}}$, v_e)
2: $a_{main} : (a_{\texttt{scope}}, a_{main}) \in \mathsf{HR}$, $a_{main} \in \mathscr{A}$
3: **for all** $x : x \in \mathsf{descendants}(a_{main}) \vee x = a_{main}$ **do**
4: $\mathsf{writes}_\tau \leftarrow \mathsf{writes}_\tau \sqcup \mathsf{writes}_\circ(x, v_e) \sqcup \mathsf{writes}_\bullet(x, v_e)$
5: **end for**
6: $\mathsf{writes}_\tau \leftarrow \left(\mathsf{poss}_\tau, \emptyset, \mathsf{inv}_\tau \cup \mathsf{dis}_\tau, \texttt{false}\right)$
7: PURGEDUPLICATESXOR(writes_τ)
8: **for all** $h_{event} : (container_{eh}, h_{event}) \in \mathsf{HR}$, $(a_{\texttt{scope}}, container_{eh}) \in \mathsf{HR}$, $h_{event} \in \mathscr{H}_{event}$ **do**
9: $a_{event} : (h_{event}, a_{event}) \in \mathsf{HR}$
10: $\mathsf{writes}_\circ(h_{event}, v_e) \leftarrow \mathsf{writes}_\tau$
11: HANDLEACTIVITY(a_{event}, v_e)
12: **end for**
13: **end procedure**

Algorithm 21 "HANDLEEVENTHANDLERS" analyzes event handlers of a `scope` activity. It is called by HANDLESUCCESSORS (Algorithm 2 on page 51) via HANDLESCOPEHANDLERS (Algorithm 19 on page 76) after the main activity contained in a `scope` has been analyzed.

Line 2 retrieves the main activity of the `scope` activity. This is necessary, as the event handlers run at any time while the main activity is active. The main activity is the only element contained in a `scope` according to HR, that is also in $\mathscr{A}$.

Line 3 iterates over all activities contained in the main activity of the `scope` activity. Different to fault handlers, this includes any activity, not only those local to the `scope` activity. The event handlers are active even if another, contained `scope` is active. Event handlers can execute at any time while the main activity is active.

Line 4 successively merges the $\mathsf{writes}_\circ$ and $\mathsf{writes}_\bullet$ writer state of each activity contained in the main activity into the temporary variable writes_τ. It is statically undecidable when an event handler runs. Because of this, the $\mathsf{writes}_\circ$ and $\mathsf{writes}_\bullet$ state of all contained activities needs to be acquired.

Line 6 moves all disabled writers to the invalid writer set. This is necessary because disabled writers at an activity are not relevant for an event handler. When the event handler runs, only the last active writer is relevant. Also, the dead path of any activity is ended by setting $\mathtt{mbd}_\tau$ to `false`. This way,

disabled writer cannot be be re-enabled at the event handler. Also, the event handler will not continue a dead path.

Line 7 calls PURGEDUPLICATESXOR (Algorithm 6 on page 61). This eliminates all duplication in the writer state writes_τ. Writers in poss_τ override writers in inv_τ. Disabled writers have been made invalid before. This is required because duplication offers (unwanted) multiple statements on dataflow, like "writer *w* might be active or not". If a writer was active at one place in the contained activity (in poss_τ) and invalid in another place (in inv_τ), then there is a possibility that the event handler can read it, so it can be moved to poss_τ.

Line 8 iterates over all event handlers of the `scope` activity. This is accomplished by finding all elements in $\mathscr{H}_{event}$ that are in the "event handler container" ($container_{eh}$) of the `scope` activity, according to HR.

Line 9 retrieves the contained activity of the event handler. It is the only element contained inside the event handler according to HR.

Line 10 copies the incoming writer state that was determined earlier to each event handler. MERGEDATA (Algorithm 3 on page 52) will be concerned with acquiring the $\mathsf{writes}_\circ$ for the activity from the handler.

Line 11 starts analysis of the activity contained inside the event handler. This is achieved by calling HANDLEACTIVITY (Algorithm 1 on page 49).

This concludes Algorithm 21.

7.11.3 Termination Handler

Algorithm 22 Analysis of termination handler

1: **procedure** HANDLETERMINATIONHANDLER($a_{\texttt{scope}}$, v_e)
2: $a_{main} : (a_{\texttt{scope}}, a_{main}) \in \mathsf{HR}, a_{main} \in \mathscr{A}$
3: **for all** $x : (x \in \mathsf{descendants}(a_{main}) \vee x = a_{main}) \wedge \mathsf{localtoscope}(x, a_{\texttt{scope}}) \wedge x \notin \mathscr{A}_{scope}$ **do**
4: $\mathsf{writes}_\tau \leftarrow \mathsf{writes}_\tau \sqcup (\mathsf{writes}_\circ(x, v_e) \text{ or } \mathsf{writes}_\bullet(x, v_e))$
5: **end for**
6: **for all** $x : (x \in \mathsf{descendants}(a_{main}) \vee x = a_{main}) \wedge \mathsf{localtoscope}(x, a_{\texttt{scope}}) \wedge x \in \mathscr{A}_{scope}$ **do**
7: $a_{term} : (h_{term}, a_{term}) \in \mathsf{HR}, (x, h_{term}) \in \mathsf{HR}, h_{term} \in \mathscr{H}_{term}$
8: $\mathsf{writes}_\tau \leftarrow \mathsf{writes}_\bullet(x, v_e)$
9: **end for**
10: $\mathsf{writes}_\tau \leftarrow \left(\mathsf{poss}_\tau, \emptyset, \mathsf{inv}_\tau \cup \mathsf{dis}_\tau, \texttt{false}\right)$
11: PURGEDUPLICATESXOR(writes_τ)
12: $h_{term} : (a_{\texttt{scope}}, h_{term}) \in \mathsf{HR}, h_{term} \in \mathscr{H}_{term}$
13: $a_{term} : (h_{term}, a_{term}) \in \mathsf{HR}$
14: $\mathsf{writes}_\circ(h_{term}, v_e) \leftarrow \mathsf{writes}_\tau$
15: HANDLEACTIVITY(a_{term}, v_e)
16: **end procedure**

Algorithm 22 on the previous page "HANDLETERMINATIONHANDLER" starts analysis of termination handlers in `scope` activities. It is called from HANDLESUCCESSORS (Algorithm 2 on page 51) via HANDLESCOPEHANDLERS (Algorithm 19 on page 76) after the main `scope` activity has been completely analyzed.

Line 2 retrieves the main activity of the `scope` activity. All writer state valid prior to the termination handler is sourced from the activities contained in the main activity.

Line 3 iterates over all activities contained in the main activity (and itself) that are local to the `scope` and not themselves a `scope` activity. $\mathsf{descendants}(a_{main})$ returns a set of all contained activities of the main activity.

The function localtoscope determines if an activity is local to the given `scope`. The first argument is the activity to examine. The second argument is the `scope` activity that the activity should be local to. localtoscope yields `false` if there is another `scope` activity contained inside the second argument's main activity (or if the main activity itself is a another `scope`) that contains the first argument. localtoscope yields `true` if the first argument is contained inside the second argument without any other `scope` activities in between, according to HR. The effect of localtoscope is that it will filter out all activities that will, on forced termination, *not* resort to the given `scope` as its termination handler. Contained `scope` activities need to be handled differently, as they have their own termination handler.

Line 4 successively merges $\mathsf{writes}_{\circ}$ *or* $\mathsf{writes}_{\bullet}$ of the activity into the temporary writes_{τ} state. The decision between $\mathsf{writes}_{\circ}$ and $\mathsf{writes}_{\bullet}$ depends on the shortness of the specified activity concerning execution time. Some activities are sufficiently short in runtime to be allowed to finish execution on forced termination, like `assign` activities. A list of these activities can be found in [OAS07, p. 135 f.].

Line 6 iterates over all `scope` activities contained in the main activity of the analyzed `scope` activity (or the main activity itself, if it is a `scope`). These `scope` activities are needed for their termination handlers, whose data will be added to the termination handler currently being analyzed.

Line 7 retrieves the contained activity of the termination handler of the `scope` in question. This is accomplished by finding the activity contained in an element both contained in the `scope` according to HR, as well as being in $\mathscr{H}_{term}$, the set of termination handlers.

Line 8 acquires the $\mathsf{writes}_{\bullet}$ writer state of each termination handler activity. This state is merged into writes_{τ}.

Line 10 moves the disabled writers to the invalid writer set. If termination occurred at an activity that was inside a dead path, the disabled writer is no longer relevant, and can be made to invalid, so as to not re-enable it later. In the same statement, the dead path is ended by setting mbd_{τ} to `false`. A dead path started before the forced termination occurred will not be continued through the termination handler. All writes in the termination handler are final with respect to overwriting prior writers, by making them invalid.

Line 11 eliminates duplication in writes_{τ}. This is accomplished by calling PURGEDUPLICATESXOR (Algorithm 6 on page 61). Writers in poss_{τ} will override writers in inv_{τ}. dis_{τ}, the set of disabled writers, has been cleared before. There might be multiple places for the termination handler to receive

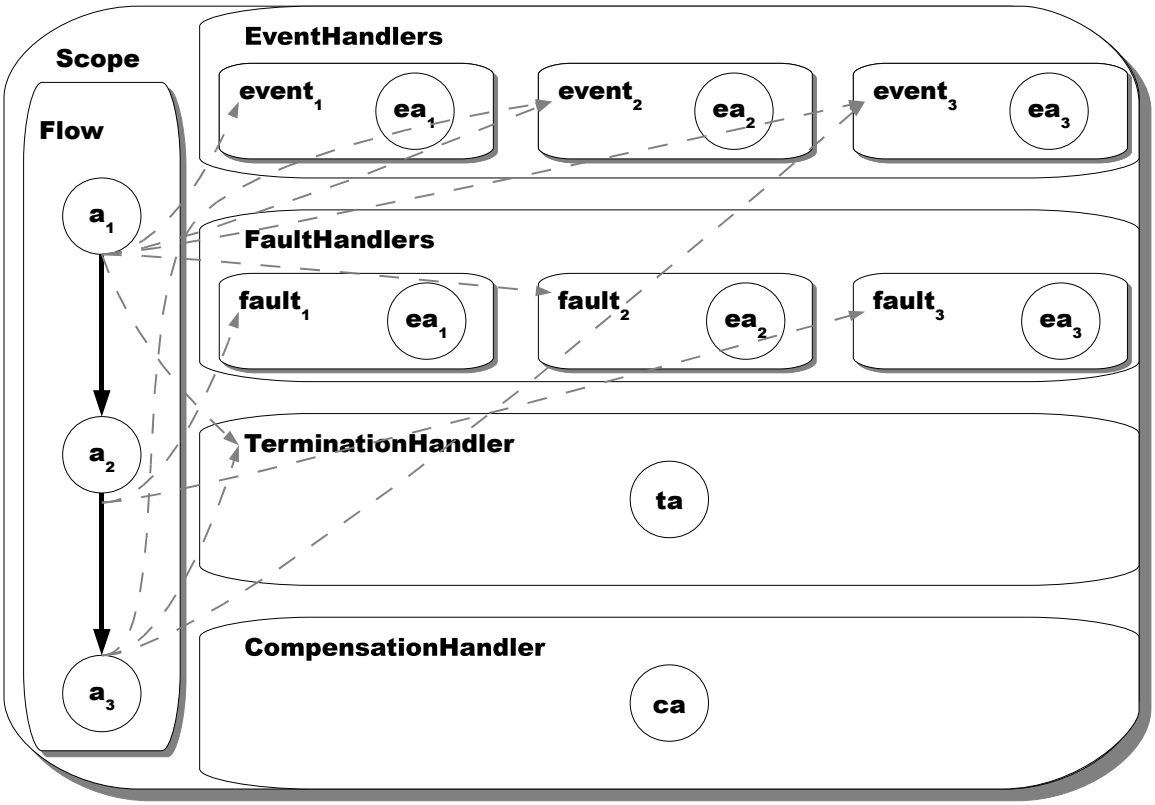


Figure 7.12: scope overview. Only exemplary data flow edges are shown.

data from. If in one of these places, a writer is a possible writer, and in another place, it is an invalid writer, then the termination handler has the possibility of reading said writer. Thus, it is valid to have poss_τ override inv_τ, which is what PURGEDUPLICATESXOR achieves.

Line 12 retrieves the termination handler of the analyzed scope activity. The termination handler is the only element that is both contained in the scope activity according to HR, as well as being in the $\mathscr{H}_{term}$ set of termination handlers.

Line 13 seeks the activity contained in the termination handler. It is the only element contained inside the termination handler, according to HR.

Line 14 copies the writer state that was determined earlier from writes_τ to $\mathsf{writes}_\circ$ of the termination handler. MERGEDATA (Algorithm 3 on page 52) will later be involved in getting the writer state to the termination handler activity.

Line 15 starts analysis on the activity contained in the termination handler by calling HANDLEACTIVITY (Algorithm 1 on page 49).

This concludes Algorithm 22.

Figure 7.12 on the previous page shows an overview of an exemplary `scope` activity after HANDLESCOPEHANDLERS has finished. When finished, fault, event and termination handlers feature data flow edges from appropriate contained activities.

7.11.4 End of `scope` Handling

Algorithm 23 Handling ending analysis of a `scope` activity

1: **procedure** HANDLEENDOFSCOPE($a_{\texttt{scope}}$, v_e)
2: $a_{main} : (a_{\texttt{scope}}, a_{main}) \in \mathsf{HR}, a_{main} \in \mathscr{A}$
3: $a_{event} : (a_{\texttt{scope}}, container_{eh}) \in \mathsf{HR}, (container_{eh}, h_{event}) \in \mathsf{HR}, h_{event} \in \mathscr{H}_{event}, (h_{event}, a_{event}) \in \mathsf{HR}$
4: $a_{fault} : (a_{\texttt{scope}}, container_{fh}) \in \mathsf{HR}, (container_{fh}, h_{fault}) \in \mathsf{HR}, h_{fault} \in \mathscr{H}_{fault}, (h_{fault}, a_{fault}) \in \mathsf{HR}$
5: $a_{term} : (a_{\texttt{scope}}, h_{term}) \in \mathsf{HR}, h_{term} \in \mathscr{H}_{term}, (h_{term}, a_{term}) \in \mathsf{HR}$
6: **if** $\mathsf{finished}(a_{main}) \wedge \forall a' \in a_{event} : \mathsf{finished}(a') \wedge \forall a' \in a_{fault} : \mathsf{finished}(a') \wedge \mathsf{finished}(a_{term})$ **then**
7: $\mathsf{writes}_\bullet(a_{\texttt{scope}}, v_e) \leftarrow \mathsf{writes}_\bullet(a_{main}, v_e) \sqcup \bigsqcup_{a' \in a_{event}} \mathsf{writes}_\bullet(a', v_e) \sqcup \bigsqcup_{a' \in a_{fault}} \mathsf{writes}_\bullet(a', v_e)$
8: $\mathsf{writes}_\bullet(s, v_e) \leftarrow \mathsf{overWrite}(\mathsf{writes}_\circ(s, v_e), \mathsf{writes}_\bullet(s, v_e))$
9: HANDLESUCCESSORS($a_{\texttt{scope}}, v_e$)
10: **end if**
11: **end procedure**

Algorithm 23 "HANDLEENDOFSCOPE" collects writer state from the main activity, and the fault and event handlers and merges it as writer state valid after the `scope`. It is called by HANDLESUCCESSORS (Algorithm 2 on page 51) when analysis of any fault, event or termination handler has finished analysis.

Line 2 determines the main activity of the `scope` activity. When the `scope` has ended, the writers of the main activity are a relevant part of the `scope`'s writers. The main activity is the only contained element according to HR that is also in the $\mathscr{A}$ set of all activities.

Line 3 retrieves the set of all event handler activities. Event handler activities are all activities that are contained in elements of type $\mathscr{H}_{event}$. These elements themselves need to be contained in the `scope` activity.

Line 4 retrieves the set of all fault handler activities. Fault handler activities are all activities that are contained in elements of type $\mathscr{H}_{fault}$. These elements themselves need to be contained in the `scope` activity.

Line 5 retrieves the termination handler activity. The termination handler activity is the only element contained in the termination handler of the `scope` activity, according to HR.

Line 6 verifies that all activities contained inside the `scope` activity have finished analysis. This includes the main activity, all event handler activities, all fault handler activities, and the termination handler activity. The finished flag is checked, as this flag is set when analysis finishes.

If not all activity analyses have ended, processing stops here. When another contained activity has finished analysis, HANDLESUCCESSORS (Algorithm 2) will call HANDLEENDOFSCOPE again.

If all activity analyses have ended, processing can continue. The writer state after each contained activity is collected and merged into $\mathsf{writes}_\bullet$ of the scope activity. Data from termination handlers does not need to be collected, as termination handlers are only executed after the forced termination invoked by an external fault handler (or forEach branching).

Line 8 updates the write state valid before the scope activity ($\mathsf{writes}_\circ$) with the writer state from inside the scope activity. This calls function overWrite (Algorithm 4 on page 56). This will also re-enable disabled writers if necessary, propagate the mbd state correctly and purge duplicate writers in the writer state.

Line 9 continues analysis after the scope activity by calling HANDLESUCCESSORS (Algorithm 2 on page 51). This will traverse outgoing links, analyze sequential successors or containing container activities.

This concludes Algorithm 23.

7.11.5 Compensation Handler

Compensation handlers have not been discussed yet. The reason is that compensation handlers can be analyzed when analyzing compensate or compensateScope activities, by which they are invoked.

Algorithm 24 on the following page "HANDLECOMPENSATE" analyzes compensation handlers when they are invoked by a compensate or compensateScope activity. It is called from HANDLEACTIVITY (Algorithm 1 on page 49) when starting analysis on a compensate or compensateScope activity.

Line 2 tests if the activity to be analyzed is a compensate activity. compensate activities trigger implicit compensation of contained scope activities.

Line 3 retrieves the next scope activity in need of compensation, and sets it on a_{scope}. There is currently no mechanism for determining the compensation order.

Line 4 is executed if the activity to be analyzed is not a compensate activity. It checks if the activity is instead a compensateScope activity.

Line 5 retrieves the target scope to be compensated, as specified in the compensateScope activity's target attribute. Now $a_{\mathtt{scope}}$ references a scope activity that has to be compensated.

Line 7 loops compensation as long as there is still a scope activity referenced by $a_{\mathtt{scope}}$. When $a_{\mathtt{scope}} = \bot$, there is no more scope activity left to compensate.

Line 8 finds the reference to the compensation handler of the scope. The compensation handler is an element contained in the scope that is also in the $\mathscr{H}_{comp}$ set.

Line 9 determines the activity contained in the compensation handler, according to HR.

Algorithm 24 Analysis of a Compensation handler

```
 1: procedure HANDLECOMPENSATE(a_comp, v_e)
 2:     if a_comp = compensate then
 3:         a_scope ← next scope to be compensated
 4:     else if a_comp = compensateScope then
 5:         a_scope ← given scope
 6:     end if
 7:     while a_scope ≠ ⊥ do
 8:         h_comp : (a_scope, h_comp) ∈ HR, h_comp ∈ ℋ_comp
 9:         a_comp,h : (h_comp, a_comp,h) ∈ HR
10:         writes∘(h_comp, v_e) ← (poss•(a_scope, v_e), ∅, inv•(a_scope, v_e), false)
11:         HANDLEACTIVITY(a_comp,h, v_e)
12:         writes∘(a_comp, v_e) ← overWrite(writes∘(a_comp, v_e), writes•(a_comp,h, v_e))
13:         if a_comp = compensate then
14:             a_scope ← next scope to be compensated
15:         else if a_comp = compensateScope then
16:             a_scope ← ⊥
17:         end if
18:     end while
19:     writes•(a_comp, v_e) ← writes∘(a_comp, v_e)
20:     HANDLESUCCESSORS(a_comp, v_e)
21: end procedure
```

Line 10 copies the writer state from the end of the `scope` analysis to the compensation handler. The compensation handler can read variables that were active when the `scope` activity ended, and they are contained in $\mathsf{writes}_\bullet$ of the `scope` activity. The `scope` activity might have been in a dead path or might have been the start of it. Because the dead path of the `scope` activity is not relevant for the compensation activity, only possible and invalid writers are copied. Disabled writers are cleared and the mbd state is set to `false`.

Line 11 starts analysis on the contained compensate handler activity. This is achieved by calling HANDLEACTIVITY (Algorithm 1 on page 49). This will also use MERGEDATA (Algorithm 3 on page 52) to retrieve the prior writers from the compensation handler h_{comp}. There are no incoming or outgoing links for a compensation handler. Thus, after this call, the compensation handler has been successfully analyzed.

Line 12 updates the writer state. The writer state prior to the compensation handler is taken from the `compensate` or `compensateScope` activity, as the writes of the compensation handler are also valid in succeeding activities. The writer state valid after the compensation handler ran is set to $\mathsf{writes}_\bullet$ of the compensation activity by HANDLEACTIVITY. A call to overWrite (Algorithm 4 on page 56) updates the writers. It also merges in disabled writers, re-enables disabled writers, purges duplicate writers and determines the $\mathsf{mbd}_\bullet$ state, as necessary. The writer state is now set to be the state prior to the compensation handlers. This is necessary in case another compensation handler needs to be analyzed.

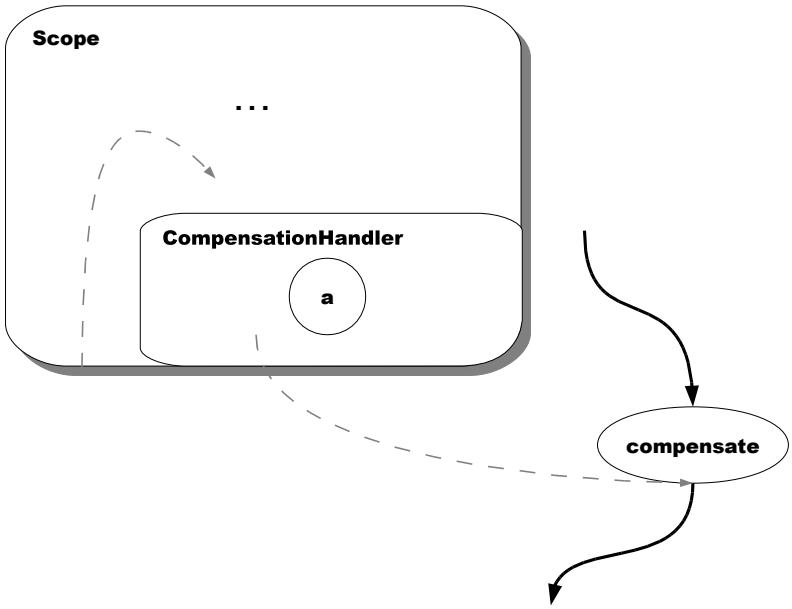


Figure 7.13: Compensation activities: `compensate` and `compensateScope` call the `compensationHandler` using $\mathsf{writes}_{\bullet}(\texttt{scope}, v_e)$

Line 13 tests if the calling activity is a `compensate` activity.

If it is, line 14 retrieves the next `scope` activity to be compensated. Again, this is necessary to determine the compensation order.

If it is not a `compensate` activity, line 15 tests if the calling activity is a `compensateScope` activity.

Line 16 sets the next `scope` to be compensated to $\perp$. `compensateScope` only compensates one single `scope` activity, the $\perp$ signifies the end of compensation analysis for this compensation activity.

Line 19 copies the writer state from $\mathsf{writes}_{\circ}$ to $\mathsf{writes}_{\bullet}$ of the compensation activity. As the most recent compensation handler analysis result is always written to $\mathsf{writes}_{\circ}$, this is a necessary step when compensation analysis ends.

Line 20 starts analysis of succeeding activities. This will analyze outgoing links, sequential successors and containing activities of the `compensate` or `compensateScope` activity.

This concludes Algorithm 24.

Figure 7.13 is a graphical display of how data flow is handled for `compensate`, `compensateScope` activities and compensation handlers.

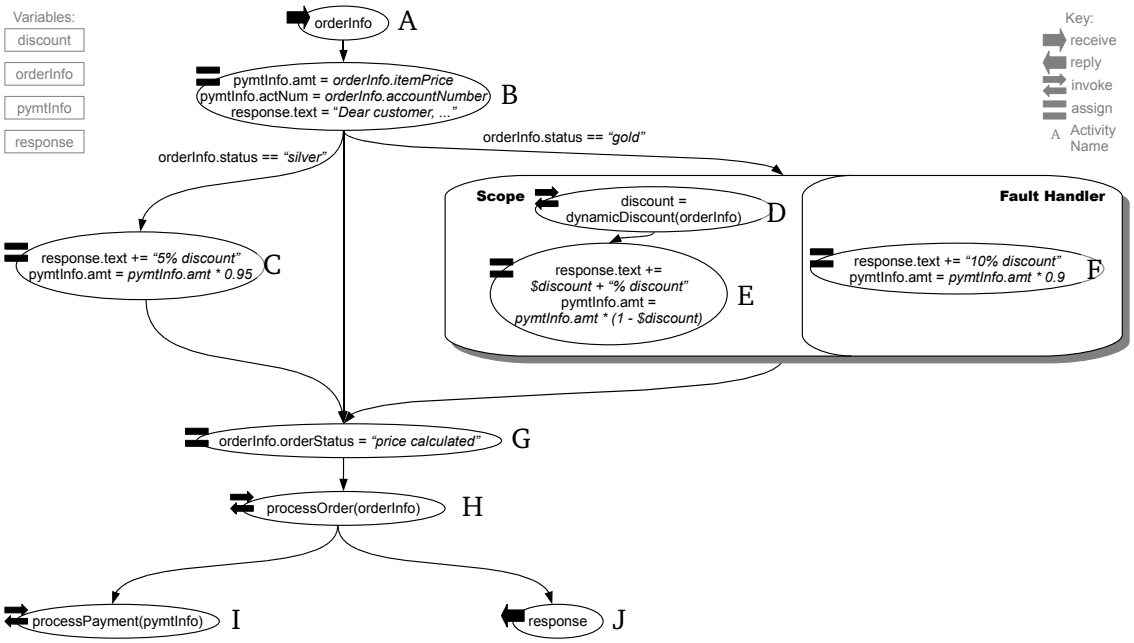


Figure 7.14: Motivating example

7.12 Revisiting the initial example

Now that the algorithm has been completely displayed and all details explained, it is time to revisit the initial example from Chapter 5. By analyzing this example, it should become clear that the algorithm works as expected.

Figure 7.14 depicts the motivating example again for easier reference. See Chapter 5 for a description of the activities.

Table 7.1 on the next page shows the $poss_{\circ}$ and $poss_{\bullet}$, $dis_{\circ}$ and $dis_{\bullet}$, $inv_{\circ}$ and $inv_{\bullet}$, $mbd_{\circ}$ and $mbd_{\bullet}$ for each activity. We begin with activity A. Both the outer `flow` activity as well as the `flow` in the `scope` activity (both not shown) are negligible, as they do not add anything to the data flow.

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
$A_\circ$	discount	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	orderInfo	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	orderInfo.orderStatus	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	pymtInfo.amt	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	pymtInfo.actNum	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\circ$	response.text	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	discount	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	orderInfo	$\{A\}$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	orderInfo.orderStatus	$\{A\}$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	pymtInfo.amt	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	pymtInfo.actNum	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$A_\bullet$	response.text	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$B_\circ$	discount	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$B_\circ$	orderInfo	$\{A\}$	$\emptyset$	$\emptyset$	`false`
$B_\circ$	orderInfo.orderStatus	$\{A\}$	$\emptyset$	$\emptyset$	`false`
$B_\circ$	pymtInfo.amt	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$B_\circ$	pymtInfo.actNum	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$B_\circ$	response.text	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$B_\bullet$	discount	$\emptyset$	$\emptyset$	$\emptyset$	`false`
$B_\bullet$	orderInfo	$\{A\}$	$\emptyset$	$\emptyset$	`false`
$B_\bullet$	orderInfo.orderStatus	$\{A\}$	$\emptyset$	$\emptyset$	`false`
$B_\bullet$	pymtInfo.amt	$\{B\}$	$\emptyset$	$\emptyset$	`false`
$B_\bullet$	pymtInfo.actNum	$\{B\}$	$\emptyset$	$\emptyset$	`false`
$B_\bullet$	response.text	$\{B\}$	$\emptyset$	$\emptyset$	`false`
$C_\circ$	discount	$\emptyset$	$\emptyset$	$\emptyset$	`true`
$C_\circ$	orderInfo	$\{A\}$	$\emptyset$	$\emptyset$	`true`
$C_\circ$	orderInfo.orderStatus	$\{A\}$	$\emptyset$	$\emptyset$	`true`
$C_\circ$	pymtInfo.amt	$\{B\}$	$\emptyset$	$\emptyset$	`true`
$C_\circ$	pymtInfo.actNum	$\{B\}$	$\emptyset$	$\emptyset$	`true`
$C_\circ$	response.text	$\{B\}$	$\emptyset$	$\emptyset$	`true`

Continued on next page

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
$C_\bullet$	discount	$\emptyset$	$\emptyset$	$\emptyset$	`true`
$C_\bullet$	orderInfo	{A}	$\emptyset$	$\emptyset$	`true`
$C_\bullet$	orderInfo.orderStatus	{A}	$\emptyset$	$\emptyset$	`true`
$C_\bullet$	pymtInfo.amt	{C}	{B}	$\emptyset$	`false`[2]
$C_\bullet$	pymtInfo.actNum	{B}	$\emptyset$	$\emptyset$	`true`
$C_\bullet$	response.text	{C}	{B}	$\emptyset$	`false`[2]
`scope`$_\circ$	discount	$\emptyset$	$\emptyset$	$\emptyset$	`true`
`scope`$_\circ$	orderInfo	{A}	$\emptyset$	$\emptyset$	`true`
`scope`$_\circ$	orderInfo.orderStatus	{A}	$\emptyset$	$\emptyset$	`true`
`scope`$_\circ$	pymtInfo.amt	{B}	$\emptyset$	$\emptyset$	`true`
`scope`$_\circ$	pymtInfo.actNum	{B}	$\emptyset$	$\emptyset$	`true`
`scope`$_\circ$	response.text	{B}	$\emptyset$	$\emptyset$	`true`
$D_\circ$	discount	$\emptyset$	$\emptyset$	$\emptyset$	`false`[3]
$D_\circ$	orderInfo	{A}	$\emptyset$	$\emptyset$	`false`[3]
$D_\circ$	orderInfo.orderStatus	{A}	$\emptyset$	$\emptyset$	`false`[3]
$D_\circ$	pymtInfo.amt	{B}	$\emptyset$	$\emptyset$	`false`[3]
$D_\circ$	pymtInfo.actNum	{B}	$\emptyset$	$\emptyset$	`false`[3]
$D_\circ$	response.text	{B}	$\emptyset$	$\emptyset$	`false`[3]
$D_\bullet$	discount	{D}	$\emptyset$	$\emptyset$	`false`
$D_\bullet$	orderInfo	{A}	$\emptyset$	$\emptyset$	`false`
$D_\bullet$	orderInfo.orderStatus	{A}	$\emptyset$	$\emptyset$	`false`
$D_\bullet$	pymtInfo.amt	{B}	$\emptyset$	$\emptyset$	`false`
$D_\bullet$	pymtInfo.actNum	{B}	$\emptyset$	$\emptyset$	`false`
$D_\bullet$	response.text	{B}	$\emptyset$	$\emptyset$	`false`
$E_\circ$	discount	{D}	$\emptyset$	$\emptyset$	`false`
$E_\circ$	orderInfo	{A}	$\emptyset$	$\emptyset$	`false`
$E_\circ$	orderInfo.orderStatus	{A}	$\emptyset$	$\emptyset$	`false`
$E_\circ$	pymtInfo.amt	{B}	$\emptyset$	$\emptyset$	`false`
$E_\circ$	pymtInfo.actNum	{B}	$\emptyset$	$\emptyset$	`false`
$E_\circ$	response.text	{B}	$\emptyset$	$\emptyset$	`false`

Continued on next page

[2] Write of C resets $\texttt{mbd}_\bullet$ to `false`

[3] Because of data flow scoping, `mbd` is now `false`

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
$E_\bullet$	discount	{D}	∅	∅	`false`
$E_\bullet$	orderInfo	{A}	∅	∅	`false`
$E_\bullet$	orderInfo.orderStatus	{A}	∅	∅	`false`
$E_\bullet$	pymtInfo.amt	{E}	∅	{B}	`false`
$E_\bullet$	pymtInfo.actNum	{B}	∅	∅	`false`
$E_\bullet$	response.text	{E}	∅	{B}	`false`
Fault handler$_\circ$[4]	discount	{D}	∅	∅	`false`
Fault handler$_\circ$[4]	orderInfo	{A}	∅	∅	`false`
Fault handler$_\circ$[4]	orderInfo.orderStatus	{A}	∅	∅	`false`
Fault handler$_\circ$[4]	pymtInfo.amt	{B,E}	∅	∅[5]	`false`
Fault handler$_\circ$[4]	pymtInfo.actNum	{B}	∅	∅	`false`
Fault handler$_\circ$[4]	response.text	{B,E}	∅	∅[5]	`false`
$F_\circ$	discount	{D}	∅	∅	`false`
$F_\circ$	orderInfo	{A}	∅	∅	`false`
$F_\circ$	orderInfo.orderStatus	{A}	∅	∅	`false`
$F_\circ$	pymtInfo.amt	{B,E}	∅	∅	`false`
$F_\circ$	pymtInfo.actNum	{B}	∅	∅	`false`
$F_\circ$	response.text	{B,E}	∅	∅	`false`
$F_\bullet$	discount	{D}	∅	∅	`false`
$F_\bullet$	orderInfo	{A}	∅	∅	`false`
$F_\bullet$	orderInfo.orderStatus	{A}	∅	∅	`false`
$F_\bullet$	pymtInfo.amt	{F}	∅	{B,E}	`false`
$F_\bullet$	pymtInfo.actNum	{B}	∅	∅	`false`
$F_\bullet$	response.text	{F}	∅	{B,E}	`false`
Fault handler$_\bullet$	discount	{D}	∅	∅	`false`
Fault handler$_\bullet$	orderInfo	{A}	∅	∅	`false`
Fault handler$_\bullet$	orderInfo.orderStatus	{A}	∅	∅	`false`
Fault handler$_\bullet$	pymtInfo.amt	{F}	∅	{B,E}	`false`
Fault handler$_\bullet$	pymtInfo.actNum	{B}	∅	∅	`false`
Fault handler$_\bullet$	response.text	{F}	∅	{B,E}	`false`

Continued on next page

[4] Fault handler receives data from all activities of main `scope` activity

[5] PURGEDUPLICATESXOR (Algorithm 6 on page 61) purges duplicates in $\mathsf{inv}_\circ$

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
`scope`•	discount	{D}	∅	∅	`false`
`scope`•	orderInfo	{A}	∅	∅	`true`[6]
`scope`•	orderInfo.orderStatus	{A}	∅	∅	`true`[6]
`scope`•	pymtInfo.amt	{E,F}	{B}	∅[7]	`false`
`scope`•	pymtInfo.actNum	{B}	∅	∅	`true`[6]
`scope`•	response.text	{E,F}	{B}	∅[7]	`false`
G◦	discount	{D}	∅	∅	`false`
G◦	orderInfo	{A}	∅	∅	`false`[8]
G◦	orderInfo.orderStatus	{A}	∅	∅	`false`[8]
G◦	pymtInfo.amt	{B,C,E,F}	∅[9]	∅	`false`
G◦	pymtInfo.actNum	{B}	∅	∅	`false`[8]
G◦	response.text	{B,C,E,F}	∅[9]	∅	`false`
G•	discount	{D}	∅	∅	`false`
G•	orderInfo	{A}	∅	∅	`false`
G•	orderInfo.orderStatus	{G}	∅	{A}	`false`
G•	pymtInfo.amt	{B,C,E,F}	∅	∅	`false`
G•	pymtInfo.actNum	{B}	∅	∅	`false`
G•	response.text	{B,C,E,F}	∅	∅	`false`
H◦	discount	{D}	∅	∅	`false`
H◦	orderInfo	{A}	∅	∅	`false`
H◦	orderInfo.orderStatus	{G}	∅	{A}	`false`
H◦	pymtInfo.amt	{B,C,E,F}	∅	∅	`false`
H◦	pymtInfo.actNum	{B}	∅	∅	`false`
H◦	response.text	{B,C,E,F}	∅	∅	`false`
H•	discount	{D}	∅	∅	`false`
H•	orderInfo	{A}	∅	∅	`false`
H•	orderInfo.orderStatus	{G}	∅	{A}	`false`
H•	pymtInfo.amt	{B,C,E,F}	∅	∅	`false`
H•	pymtInfo.actNum	{B}	∅	∅	`false`
H•	response.text	{B,C,E,F}	∅	∅	`false`

Continued on next page

[6] No write inside `scope`
[7] Duplicates B and E are removed by PURGEDUPLICATESXOR (Algorithm 6 on page 61)
[8] Returns to `false` because of enabling link B to G
[9] B is re-enabled

Activity	Variable element	possible writers	disabled writers	invalid writers	may be dead
$I_\circ$	discount	{D}	∅	∅	`false`
$I_\circ$	orderInfo	{A}	∅	∅	`false`
$I_\circ$	orderInfo.orderStatus	{G}	∅	{A}	`false`
$I_\circ$	pymtInfo.amt	{B,C,E,F}	∅	∅	`false`
$I_\circ$	pymtInfo.actNum	{B}	∅	∅	`false`
$I_\circ$	response.text	{B,C,E,F}	∅	∅	`false`
$I_\bullet$	discount	{D}	∅	∅	`false`
$I_\bullet$	orderInfo	{A}	∅	∅	`false`
$I_\bullet$	orderInfo.orderStatus	{G}	∅	{A}	`false`
$I_\bullet$	pymtInfo.amt	{B,C,E,F}	∅	∅	`false`
$I_\bullet$	pymtInfo.actNum	{B}	∅	∅	`false`
$I_\bullet$	response.text	{B,C,E,F}	∅	∅	`false`
$J_\circ$	discount	{D}	∅	∅	`false`
$J_\circ$	orderInfo	{A}	∅	∅	`false`
$J_\circ$	orderInfo.orderStatus	{G}	∅	{A}	`false`
$J_\circ$	pymtInfo.amt	{B,C,E,F}	∅	∅	`false`
$J_\circ$	pymtInfo.actNum	{B}	∅	∅	`false`
$J_\circ$	response.text	{B,C,E,F}	∅	∅	`false`
$J_\bullet$	discount	{D}	∅	∅	`false`
$J_\bullet$	orderInfo	{A}	∅	∅	`false`
$J_\bullet$	orderInfo.orderStatus	{G}	∅	{A}	`false`
$J_\bullet$	pymtInfo.amt	{B,C,E,F}	∅	∅	`false`
$J_\bullet$	pymtInfo.actNum	{B}	∅	∅	`false`
$J_\bullet$	response.text	{B,C,E,F}	∅	∅	`false`

Table 7.1: Analysis result

7.13 Conclusion

With the presented way of traversing the BPEL process, all activities will be visited at least once. As the only loops in a BPEL process are introduced through repeatable activities (`while`, `repeatUntil`, sequential `forEach`) and the analysis of these is known to end, the algorithm will also finish.

If an activity being analyzed is a writing activity, it will be added to the list of possible writers. If an activity definitely writes after another writing activity, the first writer will be moved to the invalid set. If an activity might write after another activity (possible dead path), the first writer will be moved

to the disabled set. Disabled writers will be re-enabled in all situations where the dead path ends. Both disabling and re-enabling is based on an over-approximation, because static evaluation of XPath expressions and other conditional constructs is undecidable.

While BPEL processes can be highly parallel, this algorithm runs all of the analysis sequentially.

When querying $\mathsf{writes}_{\circ}$ or $\mathsf{writes}_{\bullet}$ of an activity,

- the possible writers set contains activities that all could have written to the variable prior to the queried activity,
- the disabled writers set contains activities that were all overwritten, but might be the last writer after a dead path ends, a point at which they will be moved to the possible set again,
- the invalid writers set contains activities that were all overwritten.

To conclude, the analysis will always finish and produce a result that is an over-approximation of the actual writers at runtime.

All activities of BPEL can be analyzed. Some restrictions are necessary (see Section 4.7 on page 34), but in general, the whole WS-BPEL 2.0 semantics is supported.

Chapter 8

Implementation

Implementing the algorithm is a necessary task. It has been shown that the algorithm is simple and straightforward enough to run analysis "on paper". Yet, it is supposed to be used on larger BPEL processes also, where errors in the manual analysis would void the whole result.

For reading BPEL processes, there are libraries already at hand. Two open source libraries available to this thesis are Apache ODE and Eclipse BPEL Project.

Apache ODE[1] is a complete BPEL engine, able to execute a given BPEL process. It is written in Java, using the JDK[2] v5.0. ODE was not chosen because of greater problems compiling it and because the project architecture was harder to grasp compared to Eclipse BPEL.

The Eclipse BPEL Project[3] consists of five parts. Eclipse[4] is a framework of libraries working together to accomplish many software development tasks. One part of the BPEL Project is the BPEL designer, which aids in visually creating BPEL processes, and has already been used to create the BPEL file of the example in Chapter 5. The second part is the BPEL model, consisting of Java classes generated from an EMF[5] model. The third part is a validator working on the aforementioned model. The fourth part is a runtime framework, able to execute BPEL processes like any other BPEL engine. The fifth part is a debugging framework with support for breakpoints.

Because of the elegant structure through the use of EMF models, and the familiarity of the author with these models, Eclipse BPEL has been chosen. The model enables its user to read and analyze a BPEL process easily. BPEL activities, links and other relevant structures are transferred into Java objects at runtime. Using object methods and object variables provided by the EMF model of the Eclipse BPEL Project, it is possible to implement the whole algorithm presented into a Java application.

Additionally, the Eclipse BPEL Designer could be used to visualize the data flow after the analysis produced results.

[1] Orchestration Director Engine, `http://ode.apache.org/`
[2] Java Development Kit
[3] `http://www.eclipse.org/bpel/`
[4] `http://www.eclipse.org/`
[5] Eclipse Modeling Framework, `http://www.eclipse.org/modeling/emf/`

Two small prototype implementations have been created. One was a very early implementation using Python and the standard Python XML and XPath libraries. A number of issues could be determined by this first prototype:

- Implementation without a full BPEL model in the chosen language is very complex. Even static BPEL semantics show features not existing in pure XML, such as the scoping of variables. Analysis of these features, which is included in BPEL libraries, is additional work.
- While the pseudo-code algorithm makes the impression of being easily implemented in any language, this is not strictly true. Very basic features of the pseudo-code, such as set handling and other algebraic operators have to be implemented and hidden first. Only then is it possible to implement the algorithm itself in a manner that still provides an understandable architecture.

The prototype implemented only very basic functionality, like traversing the BPEL tree and handling `assign` activities.

The second prototype was a test application accessing features of BPEL through the Eclipse BPEL EMF model. It does not contain any part of the presented algorithm.

Due to time constraints, the full analysis application has not been developed at the time of writing this work. With the algorithm being very exact and detailed in the majority of places, the complete implementation is not supposed to pose greater problems.

Chapter 9

Summary and Outlook

This work was supposed to extend the algorithm from [KKL07c] and [KKL08] to support the complete semantics of WS-BPEL 2.0. An implementation of the complete algorithm had to be provided. Additional application scenarios had to be added and implemented.

I first gave an overview over the Business Process Execution Language. Following was an overview over data flow in general. The same chapter saw two detailing sections about the Concurrent Single Static Assignment Form and about petri nets.

Fundamentals of the algorithm have been presented. A motivating example gave insight into the motivation for the thesis. An equivalence class hierarchy that preempted the algorithm has been created. It placed BPEL activities into classes of similar behavior concerning data flow.

The data flow algorithm has been revealed completely. Each procedure and function of the algorithm saw a full "walkthrough" detailing ideas and implications of each line of code. Following the algorithm was the first application to the original motivating example.

Minutiae on the planned implementation have been discussed.

In the course of creating this work, several issues (see overview of Chapter 7) also had to be corrected. This took a majority of the available time. An implementation of the presented data flow algorithm has yet to be completed. No additional application scenario has been made.

Outlook

The data flow algorithm is currently lacking accuracy concerning XPath expressions. XPath expressions cannot be fully analyzed statically in BPEL processes, as relevant data is available at runtime only. There are works discussing static analysis of XPath expressions, most prominently [Gör08] and [Sch07].

The first, [Gör08], analyses XPath expressions in BPEL processes regarding value ranges. The second, [Sch07], abstractly interprets XPath expressions to decide on choices in the BPEL process. These works can be included to improve the decidability of XPath in the work at hand. This reduces the

amount of unnecessary data flow edges, and thus reduce the number of possible writers for any reader activity.

Another interesting area of future work includes the comparison of the algorithm to other data flow algorithms, as presented in Section 3.3 on page 25.

Finally, the most useful application is the inclusion of the data flow analysis into [KKL07a], as described in Section 1.1 on page 7.

Appendix A

Notations

Notation	Meaning
compensate, compensateScope, process	Specified BPEL activities
$\mathscr{A}$	The set of all BPEL activities
$\mathscr{A}_{basic}$	The set of all BPEL basic activities, i. e. non-container activities
$\mathscr{A}_{comp}$	The set of all compensation activities, i. e. `compensate` and `compensateScope`
$\mathscr{A}_{if}$	The set of all `if` activities
$\mathscr{A}_{flow}$	The set of all `flow` activities
$\mathscr{A}_{foreach}$	The set of all `forEach` activities
$\mathscr{A}_{pick}$	The set of all `pick` activities
$\mathscr{A}_{repeatuntil}$	The set of all `repeatUntil` activities
$\mathscr{A}_{scope}$	The set of all `scope` activities
$\mathscr{A}_{seq}$	The set of all `sequence` activities
$\mathscr{A}_{while}$	The set of all `while` activities
$\mathscr{L}_{in}(a)$	The set of all BPEL links incoming to an activity a
$\mathscr{L}_{out}(a)$	The set of all BPEL links outgoing from an activity a
HR	The Hierarchy Relation (p,a) is in HR if a is contained in p in the BPEL process
LR	The Link Relation (a_s,l,a_t) is in LR if a_s and a_t are linked by the BPEL link l
SR	The Sequence Relation (a_1,a_2) is in SR if a_1 and a_2 are succeeding activities in a `sequence`
writes(a,v_e)	Writer state tuple at activity a in analysis of variable element v_e, general usage

$\mathsf{writes}_{\circ}(a, v_e)$	$\mathsf{writes}(a, v_e)$ prior to activity
$\mathsf{writes}_{\bullet}(a, v_e)$	$\mathsf{writes}(a, v_e)$ after activity
$\mathsf{writes}_{\tau}(a, v_e)$	$\mathsf{writes}(a, v_e)$, temporary usage
$\pi_i(t)$	Returns the projection of the i^{th} component of the tuple t
$\mathsf{poss}(a, v_e)$	Possible writers set, maps to first entry of $\mathsf{writes}(a, v_e)$, general usage $\mathsf{poss}(a, v_e) = \pi_1(\mathsf{writes}(a, v_e))$
$\mathsf{dis}(a, v_e)$	Disabled writers set, maps to second entry of $\mathsf{writes}(a, v_e)$, general usage $\mathsf{dis}(a, v_e) = \pi_2(\mathsf{writes}(a, v_e))$
$\mathsf{inv}(a, v_e)$	Invalid writers set, maps to third entry of $\mathsf{writes}(a, v_e)$, general usage $\mathsf{inv}(a, v_e) = \pi_3(\mathsf{writes}(a, v_e))$
$\mathsf{mbd}(a, v_e)$	"may be dead" Boolean value, maps to fourth entry of $\mathsf{writes}(a, v_e)$, general usage $\mathsf{mbd}(a, v_e) = \pi_4(\mathsf{writes}(a, v_e))$
$\mathsf{poss}_{\circ}(a, v_e)$	$\mathsf{poss}(a, v_e)$ prior to activity $\mathsf{poss}_{\circ}(a, v_e) = \pi_1(\mathsf{writes}_{\circ}(a, v_e))$
$\mathsf{dis}_{\circ}(a, v_e)$	$\mathsf{dis}(a, v_e)$ prior to activity $\mathsf{dis}_{\circ}(a, v_e) = \pi_2(\mathsf{writes}_{\circ}(a, v_e))$
$\mathsf{inv}_{\circ}(a, v_e)$	$\mathsf{inv}(a, v_e)$ prior to activity $\mathsf{inv}_{\circ}(a, v_e) = \pi_3(\mathsf{writes}_{\circ}(a, v_e))$
$\mathsf{mbd}_{\circ}(a, v_e)$	$\mathsf{mbd}(a, v_e)$ prior to activity $\mathsf{mbd}_{\circ}(a, v_e) = \pi_4(\mathsf{writes}_{\circ}(a, v_e))$
$\mathsf{poss}_{\bullet}(a, v_e)$	$\mathsf{poss}(a, v_e)$ after activity $\mathsf{poss}_{\bullet}(a, v_e) = \pi_1(\mathsf{writes}_{\bullet}(a, v_e))$
$\mathsf{dis}_{\bullet}(a, v_e)$	$\mathsf{dis}(a, v_e)$ after activity $\mathsf{dis}_{\bullet}(a, v_e) = \pi_2(\mathsf{writes}_{\bullet}(a, v_e))$
$\mathsf{inv}_{\bullet}(a, v_e)$	$\mathsf{inv}(a, v_e)$ after activity $\mathsf{inv}_{\bullet}(a, v_e) = \pi_3(\mathsf{writes}_{\bullet}(a, v_e))$
$\mathsf{mbd}_{\bullet}(a, v_e)$	$\mathsf{mbd}(a, v_e)$ after activity $\mathsf{mbd}_{\bullet}(a, v_e) = \pi_4(\mathsf{writes}_{\bullet}(a, v_e))$
$\mathsf{poss}_{\tau}(a, v_e)$	$\mathsf{poss}(a, v_e)$, temporary usage $\mathsf{poss}_{\tau}(a, v_e) = \pi_1(\mathsf{writes}_{\tau}(a, v_e))$
$\mathsf{dis}_{\tau}(a, v_e)$	$\mathsf{dis}(a, v_e)$, temporary usage $\mathsf{dis}_{\tau}(a, v_e) = \pi_2(\mathsf{writes}_{\tau}(a, v_e))$
$\mathsf{inv}_{\tau}(a, v_e)$	$\mathsf{inv}(a, v_e)$, temporary usage $\mathsf{inv}_{\tau}(a, v_e) = \pi_3(\mathsf{writes}_{\tau}(a, v_e))$
$\mathsf{mbd}_{\tau}(a, v_e)$	$\mathsf{mbd}(a, v_e)$, temporary usage $\mathsf{mbd}_{\tau}(a, v_e) = \pi_4(\mathsf{writes}_{\tau}(a, v_e))$

a_{spec}	Activity, *spec* can be any contextual specification
p	Parent
p_s	Parent of BPEL link source activity
p_t	Parent of BPEL link target activity
p_st	Shared parent of BPEL link source and target activity
$\mathscr{H}_{event}$	Set of event handlers
$\mathscr{H}_{fault}$	Set of fault handlers
$\mathscr{H}_{term}$	Set of termination handlers
$\mathscr{H}_{pick}$	Set of handlers in a `pick` activity
$E^{Q_\mathscr{V}}$	Set of all variable elements used in the process
$E_w^{Q_\mathscr{V}}$	Set of variable elements being written to
v_e	A variable element of a BPEL variable
$\sqcup$	A binary function returning the tuple of merged sets of two tuples E. g. $(a_1,b_1,c_1,d_1) \sqcup (a_2,b_2,c_2,d_2) =$ $(a_1 \cup a_2, b_1 \cup b_2, c_1 \cup c_2, d_1 \cup d_2)$
$\mathsf{jc}(a)$	The join condition of an activity a
$\mathsf{alwaystrue}(jc)$	Returns `true` if the join condition jc is always true
$\mathsf{negateslinkstatus}(jc)$	Returns `true` if the join condition jc negates the status of its only incoming link
$\mathsf{negations}(jc)$	Returns `true` if the join condition jc contains negations
$\mathsf{localtoscope}(x, a_{\texttt{scope}})$	Returns `true` if x is local to the `scope` $a_{\texttt{scope}}$, i. e. $a_{\texttt{scope}}$ contains x and there is no `scope` activity contained in $a_{\texttt{scope}}$ that contains x
$\mathsf{children}(a)$	The set of all directly contained activities (children) of activity a, according to HR
$\mathsf{descendants}(a)$	The set of all transitively contained activities of activity a, according to HR
$\mathsf{started}(a)$	Returns `true` if analysis of activity a has started
$\mathsf{finished}(a)$	Returns `true` if analysis of activity a has finished
$\mathsf{parallel}(a)$	Returns `true` if activity a is a parallel `forEach` activity
$\mathsf{sharedparent}(s,t)$	Returns the shared parent of activities s and t, which is at least the `process` element
$w(a, v_e)$	Returns `true` if activity a writes to variable element v_e

Table A.1: Notations used

Bibliography

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. (Cited on page 21)

[Ber66] A. Bernstein. Analysis of Programs for Parallel Processing. *Electronic Computers, IEEE Transactions on*, EC-15(5):757–763, 1966. (Cited on page 35)

[BK05] F. van Breugel, M. Koshkina. Dead-Path-Elimination in BPEL4WS. In *IEEE ACSD 2005*, pp. 192–201. IEEE Computer Society, 2005. (Cited on page 34)

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4), 1991. (Cited on page 22)

[CKLW03] F. Curbera, R. Khalaf, F. Leymann, S. Weerawarana. Exception Handling in the BPEL4WS Language. In *BPM 2003*, volume 2678 of *Lecture Notes in Computer Science*, pp. 276–290. Springer, 2003. (Cited on page 34)

[God06] A. Godlinski. *Static analysis to discover communication relevant data flows in BPEL-processes*. Diplomarbeit, Friedrich-Schiller-Universität Jena, 2006. (Cited on pages 22 and 25)

[Gör08] K. Görlach. *Ein Verfahren zur abstrakten Interpretation von XPath-Ausdrucken in WS-BPEL-Prozessen*. Diplomarbeit, Humboldt-Universität zu Berlin, 2008. (Cited on pages 22, 35 and 97)

[Hei03] T. Heidinger. *Statische Analyse von BPEL4WS-Prozeßmodellen*. Studienarbeit, 2003. (Cited on pages 22 and 25)

[Kha08] R. Y. Khalaf. *Supporting Business Process Fragmentation While Maintaining Operational Semantics: A BPEL Perspective*. Doktorarbeit, Universität Stuttgart, 2008. (Cited on pages 7 and 8)

[KKL06] R. Khalaf, A. Keller, F. Leymann. Business processes for Web Services: Principles and applications. *IBM SYSTEMS JOURNAL*, 45(2):425–446, 2006. (Cited on page 7)

[KKL07a] R. Khalaf, O. Kopp, F. Leymann. Maintaining Data Dependencies Across BPEL Process Fragments. In *ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pp. 207–219. Springer, 2007. (Cited on pages 7, 8, 39 and 98)

[KKL07b] O. Kopp, R. Khalaf, F. Leymann. Determining violations of Bernstein's Criterion in WS-BPEL 2.0 processes, 2007. Unpublished Note. (Cited on page 35)

[KKL07c] O. Kopp, R. Khalaf, F. Leymann. Reaching Definitions Analysis Respecting Dead Path Elimination Semantics in BPEL Processes. Technical Report Computer Science 2007/04, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Architecture of Application Systems, 2007. (Cited on pages 8, 27, 29, 41, 47, 67 and 97)

[KKL08] O. Kopp, R. Khalaf, F. Leymann. Deriving Explicit Data Links in WS-BPEL Processes. In *IEEE SCC 2008*, pp. 367–376. 2008. (Cited on pages 27, 29, 47 and 97)

[KL06] R. Khalaf, F. Leymann. Role-based Decomposition of Business Processes using BPEL. In *IEEE ICWS 2006*, pp. 770–780. IEEE Computer Society, 2006. (Cited on pages 7 and 8)

[LA94] F. Leymann, W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 33(2):326–348, 1994. G321-5545. (Cited on page 35)

[LMP97] J. Lee, S. P. Midkiff, D. A. Padua. Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In *LCPC 1997*, volume 1366 of *Lecture Notes in Computer Science*, pp. 114–130. Springer, 1997. (Cited on page 22)

[Loh07] N. Lohmann. A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In *WSFM 2007*, volume 4937 of *Lecture Notes in Computer Science*, pp. 77–91. Springer, 2007. (Cited on pages 23 and 25)

[LVO+07] N. Lohmann, E. Verbeek, C. Ouyang, C. Stahl, W. M. P. van der Aalst. Comparing and Evaluating Petri Net Semantics for BPEL. Technical report, Eindhoven University of Technology, 2007. (Cited on page 23)

[MMG+07] S. Moser, A. Martens, K. Gorlach, W. Amme, A. Godlinski. Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis. In *IEEE SCC 2007*, pp. 98–105. IEEE Computer Society, 2007. (Cited on page 22)

[NNH99] F. Nielson, H. R. Nielson, C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999. (Cited on page 21)

[OAS07] OASIS. *Web Services Business Process Execution Language Version 2.0 – OASIS Standard*, 2007. (Cited on pages 7, 12, 17, 28, 35, 65 and 82)

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. Doktorarbeit, University of Bonn, Bonn, Germany, 1962. (In German). (Cited on page 23)

[Sch07] B. Schurr. *Analyse von XPath-Ausdrücken in BPEL-Prozessbeschreibungen*. Diplomarbeit, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2007. (Cited on pages 22 and 97)

[Tur36] A. Turing. On Computable Numbers with an Application to the Entscheidungs Problem. *Proc. London Mathematical Society*, 2(42):230–265, 1936. (Cited on page 32)

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable*

Messaging and More. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. (Cited on pages 7 and 11)

[Wes07] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag, Berlin, 2007. (Cited on page 11)

[Wor01] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1, 2001. (Cited on page 12)

All links were last followed on July 17, 2008.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Sebastian Breier)