Process interference

Beest, Nick Robbert Thierry Philippe van

Process Interference:

Automated Identification and Repair

Nick van Beest

Published by: University of Groningen
Groningen, The Netherlands

Printed by: Ipskamp Drukkers B.V.
Enschede, The Netherlands

ISBN: 978-90-367-6024-9 (book)
978-90-367-6031-7 (e-book)

RIJKSUNIVERSITEIT GRONINGEN

Process Interference: Automated Identification and Repair

Proefschrift

ter verkrijging van het doctoraat in de
Economie en Bedrijfskunde
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. E. Sterken,
in het openbaar te verdedigen op
donderdag 7 februari 2013
om 12:45 uur

door

Nick Robbert Thierry Philippe van Beest

geboren op 19 augustus 1984
te Dordrecht

Promotor:	Prof. dr. ir. J.C. Wortmann
Copromotor:	Dr. A. Lazovik
Beoordelingscommissie:	Prof. dr. ir. W.M.P. van der Aalst Prof. dr. G.B. Huitema Prof. dr. M. Weske

Contents

List of Figures

List of Tables

CHAPTER 1

Research context

1.1 Introduction

Business processes constitute the core of the operational systems in organizations. Business processes typically comprise interrelated tasks executed by operational functions like purchasing, manufacturing, planning, marketing and sales. As a consequence of the complexity of modern business processes, support of these business processes by enterprise information systems (EIS) is a necessity (Dewett and Jones, 2001).These systems offer support for the work of employees. This support may consist of a wide array of functionality, including business process management, which guarantees correct sequencing of business process tasks. Due to rapidly changing environments, new stakeholders and competitors, it is important that a business process can be changed and flexibility is essential (Moitra and Ganesh, 2005).

Current organizations are characterized by long-running complicated business processes that involve many different stakeholders. If some activities or subprocesses are executed by one or more external parties, business processes are called *distributed*. Increasingly, business processes have a distributed nature. Similarly, data resources used by the business process are not necessarily proprietary to one organization and can be shared with other stakeholders. Data needed during execution of a process may also be called *distributed* if these data are owned by more than one party. In the case of data, it can be observed in reality that data are increasingly distributed as well.

Due to their distributed nature, modern business processes are required to be able to execute independent from other processes, in order to avoid an abundance of dependencies between different stakeholders. Consequently, business processes and their data are designed with an inherent assumption of independence of processes and, as a result, their data.

The representation of a single execution of a process is called a *process instance* (WfMC, 1999; Russell et al., 2005). Although, multiple instances of a process may run simultaneously, each of these is assumed to be independent (Russell et al., 2005). Accordingly, if instances of processes are executed concurrently, it is implicitly assumed by their designers that these instances cannot affect each other. However, multiple instances may require the same data over a certain timeframe. As a consequence, unanticipated interaction may arise between processes as a result of shared data usage between process instances of concurrent processes. Data, modified by an external process, may result in unexpected behaviour and undesirable business outcomes.

Example 1.1.1 (Example of interference)**.** *A customer of an energy company may decide to change his energy provider. During the execution of this process (which may take up to 6 months, depending on the contract ending date), the customer decides to move to another home. After the customers address has been changed, the process responsible for handling the switch of the energy provider may use the outdated address. As a result, there may be a discrepancy between the actual address of the customer and the address that is used for his invoices.*

A more subtle example is illustrated by Example 1.1.2, where disruptions are caused by implicit data interdependencies with the data that is modified. This interdependence of data is hard to pinpoint and is not automatically captured by data-flow analyses.

Example 1.1.2 (Example of an implicit data interdependency)**.** *Consider a business process for issuing a wheelchair for disabled people: in the Netherlands, it takes up to 6 weeks from sending the initial request to receiving an actual wheelchair. After the request, first a home visit has to take place at the patient, followed by the acquirement of requirements. Subsequently, the order is sent to the supplier, where the wheelchair is manufactured. Finally, the wheelchair is delivered to the patient.*

If in the meantime the patient has moved to a different place, it is possible that the requirements for wheelchair need to be changed. This is caused by an implicit data interdependency between address and wheelchair-order: a change of address implies that the previously executed home visit is no longer useful, as it concerned the previous home of the patient. As the requirements are partially based on the result of the (now outdated) home visit, a new home visit is necessary. The newly executed home visit may, in turn, result in a change of requirements (e.g. different dimensions due to smaller doorways). If requirements are indeed to be changed, this has a consequence for the order itself and the supplier should be notified.

Example 1.1.2 clearly shows the effect of implicit data interdependencies. Only at the delivery, the address is explicitly required. However, it is implicitly required for the order, as the order is based on requirements that are partially resulting from the home visit, which is executed at a certain address.

The process environments described in Example 1.1.1 and 1.1.2 lead to those problems that are initially not necessarily experienced inside the organization, as no error messages like a dynamic deadlock detected are signaled. Although such data interdependencies may in the worst case scenario cause process instances to fail, in most cases the regular finish of the business process does not visibly affect the performance parameters that are monitored (e.g. the number of rejected or unfinished cases). The disruption has, however, a considerable effect as the final result is undesirable from a business perspective. That is, customer satisfaction is negatively affected in the long run. The customer is seen here as an external resource involved in the execution of the concurrent business processes, which is spread over more organizations.

These problems are referred to as *process interference* (Xiao and Urban, 2007; Van Beest et al., 2010a). Process interference is defined in this thesis as the situation where data modifications by one process affect one or more other concurrently executing processes, which potentially causes an undesired process outcome for one or more of these processes. More specifically, consider a process P where some data element d is read by process P and a part of the subsequent execution of P assumes that d remains unchanged. A process Q interferes with P if there is a sequence of events where d is modified by process Q, while process P is still in the part of execution where d is presumably unchanged. A formal definition of process interference is provided in Section 4.1.2.

1.1.1 Generic interference example

Heretofore, process interference has been primarily illustrated by means of an example. In this section, a more generic description of process interference will be provided. In Figure 1.1, two independent concurrent processes are shown using a common database. That is, a mutation of a data element by one process, affects the value of the corresponding data element of the other process as well.

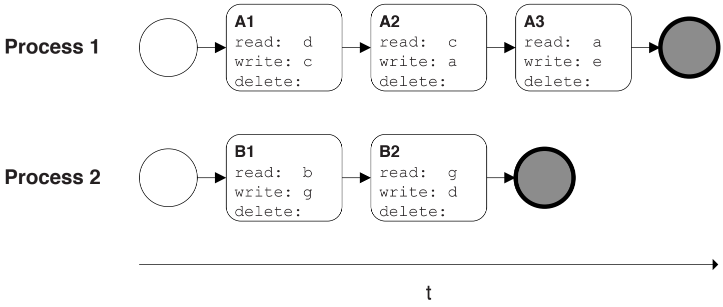


Figure 1.1: Business process with concurrent data change.

Activity A1 reads d and writes to c. Subsequently, activity A2 reads the value of c and writes to a accordingly. Finally, activity A3 reads a and writes to e. Implicitly, the value of e is by transitivity dependent on d. If a concurrent process changes that value of d (activity B2) *after* it has been read by process 1, potentially e will have the wrong value assigned.

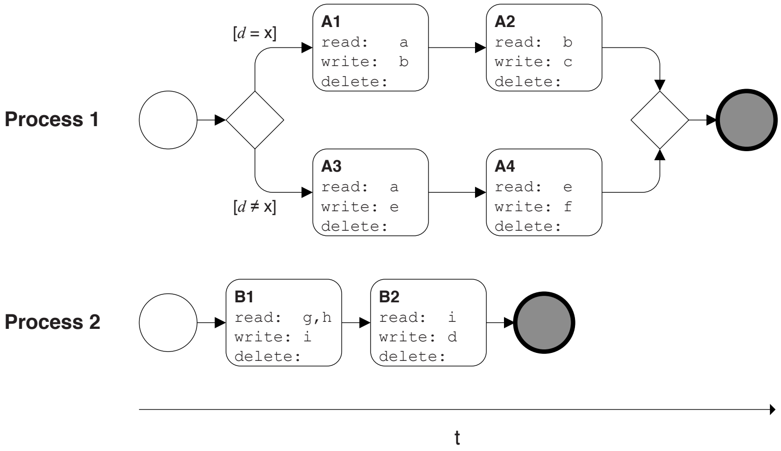


Figure 1.2: Conditional branches with concurrent data change.

Similarly, such an external data mutation may also affect the subprocesses that follow after the evaluation of a condition. In Figure 1.2, for example, the decision is based on the value of d. That specific decision determines whether A1 and A2 are executed or A3 and A4. If d is changed by another process during execution of A2, this may have consequences for the correctness of the activities being executed at that time. That is, as a result of the data change, currently the wrong branch of activities is executed.

In Chapter 4, a formal definition of process interference will be provided.

1.1.2 Verification

The *control-flow* of a process describes the execution order of activities through different constructs, e.g. sequence, choice, parallelism and join synchronization (Van Der Aalst et al., 2003a). In order to ensure soundness of a business process, much research has been done concerning verification of processes. This so-called *workflow verification* checks the control-flow of the process to guard for e.g. deadlocks and livelocks. Although all organizational processes inherently use data, data is in most cases seen as a black box in workflow verification techniques. However, the link between data and processes requires data to be a fundamental part of workflow verification. In theory, verification techniques can be extended to data as long as the domain is finite. For instance, CPN Tools (Jensen and Kristensen, 2009) can model processes with data, allowing for model checking on the resulting finite state space. However, existing approaches do not manage huge or infinite data domains very well. That is, there are existing approaches that work, but they only work when data domains are manageable.

Such a data extension has been proposed by Trčka et al. (2009), by extending the control-flow model with data elements (workflow nets with data), which allows for capturing both control-flow and data-flow errors. Sidorova et al. (2011) present an extension of workflow with data operations, in order to provide a precise analysis of the soundness of a workflow. In Monakova et al. (2009), a verification algorithm is presented to verify business constraints in the process. However, these verification techniques analyze workflows and their data-flow in isolation. That is, data dependencies between concurrent processes are only visible when the processes are modelled in the same domain. More specifically, data changes by activities from a different business process are ignored. Due to the lack of attention to data changes by other processes during execution, process interference may still occur. In addition, data resources are increasingly shared with other external actors and processes, where a part of the process to be verified is defined and implemented outside the organizational boundaries. This implies that all data changes initiated by processes outside the scope of the process model are not checked and cannot be verified. As a result, identification of this problem is rather complex. However, verification techniques can be used for detection of potential problems (as shown in Chapter 4). In Chapter 4, we will adapt this technique to provide a structured definition of the problem described here.

Although the analysis of data dependencies and process interference itself is investigated in academic literature (see e.g. (Xiao and Urban, 2008)), the presented methods apply only in context of failing processes and refer to highly distributed environments or service-oriented environments. That is, the provided solution is situation and implementation dependent. In practice, however, these interfering processes do not necessarily fail. Rather, they may execute correctly (i.e. without internal error messages) but provide the wrong business result, especially from a customer perspective. Furthermore, process interference is not limited to distributed or service-oriented environments. In addition to failing processes, Urban (Urban et al., 2011) proposes an approach to define (design-time) rules to specify the required compensation actions in case of interference, incorporating events like exceptional conditions or unavailable activities. Nevertheless, problems occurring at a regularly executing process due to the use of inaccurate data are not considered.

1.2 Problem statement

Process interference occurs far more often than most people realize. Processes are developed under the assumption that case-related data are stable, and this assumption is in general not true. As soon as case-related data are changed, processes may yield wrong results, however, without leading to immediate software errors. Because there is often not an immediate software error, the incorrect impression exists that the process runs well. These errors in the real world lead to customer complaints, legal cases, and many untraceable societal costs (Van Beest et al., 2010b). However, their root cause, process interference, is overlooked in process management software architectures.

In addition, these situations are not limited to those processes, which include choice and parallelism, but also appear when multiple sequential processes are executed concurrently. Furthermore, this problem is not necessarily related to a specific technology used; this problem is independent from technical implementation details.

This current lack of existing mechanisms to manage interdependencies and interference stems from the complexity of the problem itself and, more importantly, the troublesome identification of interference. More specifically, the undesirable business process outcomes are characterized by a rather regular end of the business process with only small internal disruptions to the organization. As a result, these problems are initially not necessarily experienced inside the organization, as no error messages like a dynamic deadlock detected are signaled. However, the external part of the disruption has a considerable effect, as the problem induced by interference is primarily noticed by the external stakeholders (mostly customers). As a result, full identification of the individual troublesome cases and the severity of interference throughout the entire process is rather complex.

The problem is, therefore, twofold. First of all, the identification is difficult, due to the complexity of the problem and magnitude of the state space of the problem. Second, once process interference is identified, it is difficult to prevent due to external stakeholders, volatile data, and lack of model checking and verification possibilities other than checking each individual interference case.

Similarly, this thesis will be divided into two parts. The first part will address the identification of business process interference. Consequently, the following research questions are addressed in the first part of this thesis:

Research Question 1

How can business process interference be identified?

Research Question 2

How can the severity of existing business process interference be assessed?

The second part of this thesis will describe the design of a solution to business process interference. Therefore, the following research questions are addressed in the second part of this thesis:

Research Question 3

How can business process interference be prevented in enterprise information systems and which artefacts are required to ensure process and data consistency?

Research Question 4

What techniques are required for automated recovery from process interference?

1.3 Methodology

This research is triggered by a business problem that is recognized by both organizations and their stakeholders, such as customers. Current artifacts are insuffient to overcome the business problems as described. This lack of suitable existing artifacts prevents routine design and requires the design of a new (set of) generalizable artifact(s) to resolve this business problem.

The design science research methodology (DSRM) process model as described by Peffers et al. (2007) provides a clear description of the design process and describes the steps from problem definition to evaluation and communication in detail. In the problem identification and motivation step, the specific research problem is defined and the value of a solution for that problem is justified. In the second step the objectives are defined for a solution. The objectives are inferred from the problem definition. The third step comprises the actual design of the new artifacts, which can be in the form of constructs, models, methods, or instantiations (March and Smith, 1995; Hevner et al., 2004). After the design, the artifacts are demonstrated by solving one or more instances of the problem. The fifth step concerns the evaluation of the design, by testing the effectiveness of the artifacts to solve the business problem under investigation. In the final step, the gathered knowledge from the problem and the designed artifacts are communicated to the appropriate audience and added to the knowledge base (Peffers et al., 2007).

As the research is triggered by the observation of the business problem, the research presented in this thesis can be categorized as problem initiated design science, according to the DSRM model (Peffers et al., 2007) shown in Figure 1.3.

Due to the problem-solving nature of design-science, an explicit generalization step

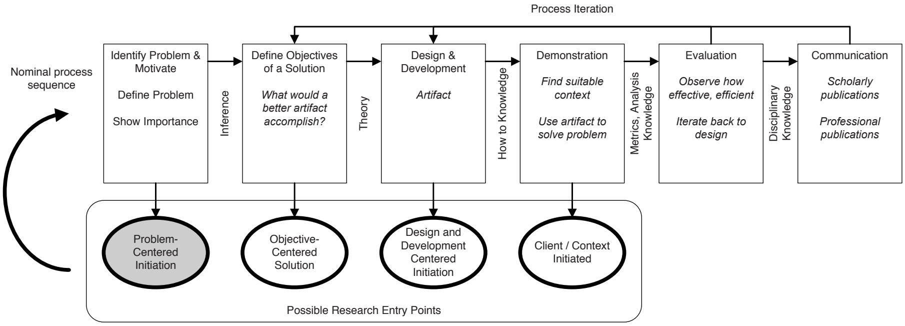

Process Iteration
Nominal process sequence
Identify Problem & Motivate
Define Problem
Show Importance
Inference
Define Objectives of a Solution
What would a better artifact accomplish?
Theory
Design & Development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artifact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary Knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design and Development Centered Initiation
Client / Context Initiated
Possible Research Entry Points

Figure 1.3: Design Science Research Methodology process model (Source: (Peffers et al., 2007))

to identify the generalized problem absent or not elaborated (including the model of Peffers et al. (2007)). Consequently, the resulting solution is context-specific. Although the regulative cycle (Van Strien, 1997) contains a diagnosis step to investigate the detailed underlying causes of the problem, each cycle is initiated by a context-specific problem, which is intended to be resolved by the design. Correspondingly, the frameworks lack an explicit validation step to ensure the applicability of the solution to all organizations where the problem occurs. The model of Peffers, for example, only evaluates the efficiency and effectiveness of the particular solution. As a result, it is inconclusive whether a designed solution would apply to all organizations with a conceptually similar problem. However, this thesis does not intend to limit the provided solution to a single instance, but rather provides a generic solution applicable to all process interference cases.

The problem that is investigated in this thesis exists on a conceptual level, regardless of the technologies used, and pertains to many organizations. A specific solution for a particular situation with a particular implementation would, therefore, not suffice. The independence from implementation suggests that the problem itself has a more generic nature. That is, the problems as perceived by organizations and their stakeholders are merely symptoms of an underlying, more fundamental problem. In this thesis, the artifacts are designed in a way that they resolve the fundamental problem rather than the mere symptoms, in order to provide a generalizable solution to the symptoms as identified by organizations. Therefore, the methodology used in this research comprises two additional steps over the existing design science research methodology developed by Peffers et al. (2007). The first step that is added to the DSRM model concerns the identification of the generalized problem. The second additional step concerns the validation of the design with a case, as the design process should be connected with the application context (Hevner, 2007). A solution for this generalized problem leads to generalized academic knowledge. This step is necessary for corroboration of the entire work, from problem analysis via generalization to the designed solution.

In Figure 1.4, a detailed overview is provided of the research methodology used in this thesis. This thesis consists of three parts. The first part is concerned with the analysis of the problem in the business context. An in-depth analysis of the problem will be performed, by conducting two case studies to identify process interference based on detailed documentation about the process and experience of users. Moreover it comprises the additional generalization step, to formulate a more formal generic problem. In addition to the case studies, the generic problem is also derived from existing experience and knowledge as well as the existing knowledge and formalisms available in literature.

The second part of the thesis comprises the design process. Based on the generalized problem, a set of new modeling constructs is developed during an iterative design process. Furthermore, the design is grounded in the existing literature, in order to ensure a fit with existing business process modeling paradigms. The newly designed artifacts are the primary addition provided by this research to the academic knowledge base. Finally, the design constructs are implemented as an extension to an existing business process modeling tool.

The third part of the thesis comprises the evaluation of the design and the validation. After implementation of the design, the artifacts are applied to another real-life case. The case used for the assessment of the artifacts with the requirements is independent from the cases used in the problem definition phase. This independence of cases is necessary, in order to ensure that the designed artifacts are not biased towards the symptoms as identified from the environment. A representative business process of this case is modelled including the newly designed constructs.

Next, a simulation of the process is executed with disruptions, in order to test and evaluate the effectiveness of the designed concepts.

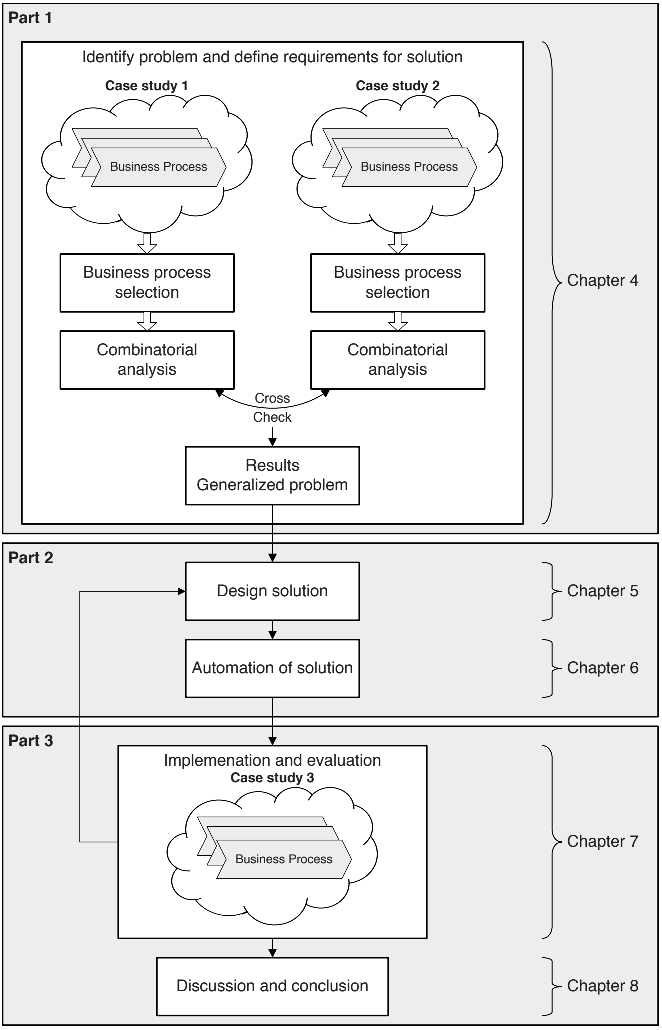


Figure 1.4: Research methodology

Throughout this research, three case studies have been conducted, serving different purposes. An overview of these cases is provided in Table 1.1.

Case	Purpose	RQ	Thesis part
1. Energy supply chain	Problem identification, development of process interference identification methodology.	1 and 2	1
2. Telecom company	Problem identification, development of process interference identification methodology.	1 and 2	1
3. Dutch local government	Testing and validation of designed solution.	3 and 4	2 and 3

Table 1.1: Overview of used cases

Although each case concerns a different industry, the selected cases share the following important properties:

1. Complex and long-running processes.
2. Parallel execution of processes that use implicitly related data.
3. Multiple stakeholders.

This similarity of properties of the investigated cases implies that all cases are liable to the problems as described. As a result, all cases are very well suitable for both identification of the generic problem and evaluation of the designed solution. Furthermore, the separation between Part 1, Part 2 and Part 3 (and the subsequent use of different cases) prevents the designed artifacts from being biased towards specific properties or symptoms of the cases used for problem identification.

1.4 Thesis structure

The remainder of this thesis is structured as follows. Chapter 2 provides the state of the art in business process modelling, verification and business process reconfiguration. In Chapter 3, a detailed overview of the case studies is provided, along with a description of the data collection and the processes under investigation. In Chapter 4, a method is presented along with an operational tool that enables to identify the potential interference and analyze the severity of the interference resulting from concurrently executed processes.

In the second part of the thesis, the design of the artifacts is provided in Chapter 5 and Chapter 6. Chapter 5 introduces the concept of dependency scopes to represent the dependencies between processes and data sources. In addition, intervention processes are developed to repair inconsistencies using dynamic reconfiguration during execution of the process. In Chapter 6, an approach is presented for automating the generation of intervention processes at runtime, by using domain-independent AI planning techniques. Furthermore, an algorithm is presented for automating the specification of dependency scopes.

Finally, the third part of the thesis, which is covered by Chapter 7, describes the implementation and evaluation of the design by means of case study 3. A prototype of the architecture (as presented in Chapter 6) is implemented and evaluated. The aim of the evaluation is to demonstrate the effectiveness of the approach with respect to the working example presented in Section 7.1 and to test the performance with respect to the time that is required to generate the required intervention processes.

Chapter 8 provides a summary of the work and provides a detailed discussion on each of the developed concepts.

In Appendix A, the low-level results of the analysis are provided, including the meta-data as used by the analysis tool. In Appendix B, a complete BPEL represention is provided of the eGovernment process. The variable interdependencies of that process are specified in Appendix C. The AI Planning representation of the eGovernment process is provided in Appendix D.

1.5 Publications in this thesis

The work has been developed in collaboration with various people (as the publications indicate), in particular with Hans Wortmann, Alexander Lazovik, Eirini Kaldeli and Pavel Bulanov. The work presented in the thesis is primarily concerned with the problems regarding process interference in organizations. We have developed a method to identify the extent of process interference based on process documentation [3][5] (Chapter 4). Using simulation, troublesome cases can be identified and the severity of the interference can be determined. Furthermore, we have developed a number of concepts to resolve process interference by runtime reconfiguration [4] (Chapter 5). Consistency of business processes can be restored by runtime generation of intervention processes [1], based on automated discovery of dependency scopes [2] (Chapter 6) and AI planning techniques, which have been developed by Eirini Kaldeli. A prototype has been implemented (in close collaboration with Pavel Bulanov) and evaluated in [1] and [2] (Chapter 7).

[1] Van Beest, N.R.T.P., Kaldeli, E., Bulanov, P. Wortmann, J.C., Lazovik, A., 2012. Automated Runtime Repair of Business Processes. Submitted.

[2] Van Beest, N.R.T.P., Kaldeli, E., Bulanov, P. Wortmann, J.C., Lazovik, A., 2012. Automatic Detection of Business Process Interference. International Workshop on Knowledge-intensive Business Processes (KiBP'12), Rome, Italy. Invited paper.

[3] Van Beest, N.R.T.P., Lazovik, A., Wortmann, J.C. Automated discovery of business process interference. In progress.

[4] Van Beest, N.R.T.P., Bulanov, P. Wortmann, J.C., Lazovik, A., 2010. Resolving Business Process Interference via Dynamic Reconfiguration. 8th International Conference on Service Oriented Computing (ICSOC-2010), Lecture Notes in Computer Science, Vol. 6470/2010, pp 47-60.

[5] Van Beest, N.R.T.P., Szirbik, N.B., Wortmann, J.C., 2010. Assessing The Interference In Concurrent Business Processes. Proceedings of the 12th International Conference on Enterprise Information Systems, ICEIS 2010, Springer, Vol. 3, pp 261-270.

[6] Van Beest, N.R.T.P., Szirbik, N.B., Wortmann, J.C., 2009. A Vision For Agile Model-driven Enterprise Information Systems. Proceedings of the 11th International Conference on Enterprise Information Systems, ICEIS 2009, Springer, pp 188-193.

CHAPTER 2

Related work

2.1 Business process modelling

2.1.1 Process representation

In the early 1990s, business process (BP) modelling emerged with the purpose of analyzing the BP. BP models are capable of specifying the activities in the BP and the control-flow between these activities. The modeling of business processes has become a strategic goal in many organizations (Weske et al., 2006) and the graphical representation of these models proved to be useful to determine potential areas of improvements, forming the basis of BP redesign (Davenport and Short, 1990). Along with the appearance of integrated information systems, BP modelling also gained popularity for design and specification of enterprise information systems (Johannesson and Perjons, 2001).

When BPs are graphically specified with the purpose of IS design, two categories of BPs can be identified. One category is mainly consultancy oriented, i.e. the models are primarily used among business analysts and system architects as a graphical language for specifying the business process. For instance, the Unified Modeling Language (UML) provides a set of graphical modeling notations to model an IS. UML is designed in the 90s and has been a widespread standard in software engineering since 1997 (Booch et al., 2005; Fowler and Scott, 2000). In UML, the BP is modelled by means of activity diagrams, where activities are connected through arcs. In 2005, UML has been updated to version 2.0, to support the specification of pre and post conditions, events and actions, time triggers, time events, and exceptions (OMG, 2005).

Similarly, the Business Process Management Initiative introduced the Business Process Modelling Notation (BPMN) in 2004, which has the primary goal to provide a notation that is readily understandable by all business users (White, 2004). The process flow is represented in a graph-oriented way, where the explicit control-flow is defined by events, activities, and gateways, which are connected through sequence flows and message flows (Kopp et al., 2008; White, 2004). In 2009, BPMN was updated to version 2.0 (OMG, 2009). Although BPMN 2.0 includes detailed execution semantics for all BPMN elements, it still only provides an informal description of those semantics (OMG, 2009).

Although intuitively readable, neither UML or BPMN are formal, executable modelling languages (Urban et al., 2011; Kopp et al., 2008). Due to the lack of formal semantics, execution based on these models is not directly possible. However, in many cases these models can be converted into a models based on a genuine BP formalism in order realize such an executable model as shown by Ouyang et al. (2006) and Dijkman et al. (2008).

The other category is focused on formal activity sequencing and coordination (i.e. the control flow perspective), using Petri-nets (Van Der Aalst, 1998), activity-based workflow modeling (Bi and Zhao, 2003) or block-structured modelling (e.g. Business Process Execution Language (BPEL), (Juric, 2006)). These formal BP specifications allow for automated analysis of the model, in order to discover syntactic errors, deadlocks, livelocks and orphan activities through modeling and analysis (Van Der Aalst, 1997; Trčka et al., 2009). These models can be used as an executable specification to be used by the IS.

For instance, Workflow Nets (WfNs) provide a formal basis for workflow modelling and offer the possibility for model-checking and verification (Van Der Aalst, 1997; Verbeek et al., 2001). Model checking is a set of formal techniques that is used to verify systems against its specifications (Clarke et al., 1999). Workflow modelling has its roots in Petri Nets, which were invented in 1962 by Carl Petri (Petri, 1962). A Petri Net is a directed bipartite graph with nodes representing either places or transitions, which are connected through directed arcs [1].

The Business Process Execution Language for Web Services (BPEL) (Arkin et al., 2007) is a block-structured language, where control flow is defined similar to existing programming languages by using block-structures such as if or while. It is considered the de-facto standard for implementing BPs on top of web services technology (Verbeek, 2005; Weske et al., 2006; Ouyang et al., 2007) and has been designed specifically to support web services-based processes as an important part of an SOA. Similar to WfNs, BPEL models provide the possibility for model-checking and verification as well, e.g. using conversion to Petri Nets (Ouyang et al., 2007) or pi calculus (Liu et al., 2007).

2.1.2 Data representation

The information used by business processes is represented by data, which is stored in one or more databases. A database is usually structured according to a data model (or ontology), which describes the semantics of the data. The data model may contain rules and constraints to ensure the data to be consistent and to correspond with the reality in the business environment (Nicolas, 1982; Alwan et al., 2011).

Every activity in a business process requires data for its execution. Data can be read, new data can be generated or existing data can be modified. A decision (e.g. an XOR split) requires data to select the consecutive execution path (Meda et al., 2010). As a result of these interactions between activities and data, data itself may be continuously changing during runtime. Consequently, design-time checking and verification alone is not enough to ensure data consistency. Therefore, runtime

[1] For a review of the history of Petri nets and an extensive bibliography, the reader is referred to Murata (1989).

consistency checking and management of *data transactions* is required to maintain data integrity (Bernstein et al., 1987).

A *transaction* is a set of operations on data by a database in a reliable way independent of other transactions. Reliable execution of data transactions is ensured by four basic properties. First of all, transactions are *atomic*. That is, transactions are executed indivisibly (Haerder and Reuter, 1983). Secondly, any mutation to the database is achieved through data transactions. This implies that every correct transaction, committing its results, brings the database from one consistent state into the other, thus preserving *consistency* (Haerder and Reuter, 1983). Thirdly, each transaction is unaware of any other concurrently executed transactions. That is, events within a transaction are hidden to other transactions, which is referred to as *isolation*. Finally, once a transaction is committed, it cannot be withdraw and is, therefore, final and guaranteed to survive any system failures *durable*.

The simultaneous enactment of various process instances implies a concurrent execution of database transactions. Concurrent execution may result in data interference as a result of interleaving transactions. Database theory has provided several solutions in the past to manage concurrent transactions, in order to preserve data consistency. Preserving consistency and achieving isolation is referred to as concurrency control. An overview can be found in Bernstein et al. (1987).

Consistency is expected to hold not only for individual transactions but also when a set of transactions completes. Methods for ensuring consistency during completion of a set of transactions are presented by, for example, Garcia-Molina and Salem (1987) and Korth and Speegle (1988). An overview of database consistency requirements and transaction correctness properties is presented by Ramamritham and Chrysanthis (1993).

2.2 Interference

For a single program without parallelism, techniques exist to ensure that consistency of the data is maintained during execution. However, consistency problems may occur if independent processes access and change the same data without global coordination via e.g. a database management system. (this possibility became obvious when the database management systems of the past started to have multiple, concurrent access). In the study of such systems for classic databases of transaction-based systems, the focus has been primarily on implementing ACID transaction semantics (for a review, see (Xiao et al., 2006)). This tradition has been taken into account in the more recent service composition research. This research investigates technology to avoid data consistency problems when designing composite services. In Xiao et al. (2006), a global database of object history execution (as the PHCS process history capture system) is proposed, which is appropriate in a dynamic service composition environment, with frequent rollbacks and cascading compensated activities. Nonetheless, from the perspective of the design-time data-flow analysis, there have been only a few recent research approaches (Xiao and Urban, 2007; Meda et al., 2010; Trčka et al., 2009) to provide a systematical discovery of data-flow errors in business processes. As observed by Sun et al. (2006), existing commercial workflow systems, for example, do not yet provide adequate tools for data-flow analysis at design time.

Concurrent processes and their instances are assumed to be independent. Although temporal analysis methods exist to verify resource constraints, these methods assume coordinated concurrent execution. In (Li and Yang, 2005) for instance, a formal approach for dynamic verification of temporal constraints is proposed. In (Trčka et al., 2009), temporal logic is used for data-flow analysis in business processes to ensure soundness of both the control-flow and the data-flow (Trčka et al., 2009). In Sidorova et al. (2011) an extension of workflow with data operations provided, in order to provide a precise analysis of the soundness of a workflow. In Monakova et al. (2009), an algorithm is presented extending BPEL process verification with a data-flow analysis. In (Sun et al., 2006), a data-flow matrix is proposed to integrate data-flow models in the control-flow (or workflow) model, in an attempt to detect data-flow errors, redundant data, and potential data conflicts.

In distributed environments, multiple non-synchronized processes are executed concurrently *within* an organization or *between* organizations, especially when parts of the information system are delegated or outsourced (as described by Balsters and Huitema (2007)). As a result, data can be changed by another process having simultaneous access to a distributed database. In addition, data can also be changed in reality without awareness of the currently executing process. That is, there may be a mismatch between data in reality and data as assumed the process. In both cases, the data used by the process is different than the data outside the system (i.e. reality or the distributed database). The situation where a data change in reality is not taken into account during execution of a process instance is referred to as an *external data change*.

Consequently, traditional verification techniques for workflow and data-flow are not sufficient for ensuring the correctness of such BPs, as they assume that process and data interactions are available and can be predefined in advance. However, not all interactions are known or pre-specified, since data can be changed externally, without providing a notification to the business process in progress. As a result, runtime disruptions due to external data changes cannot be prevented or avoided.

Ensuring consistency between the internal data representation and the external business reality is more complex and cannot be easily resolved by these transaction correctness properties. It requires a continuous observation of reality and a comparison with the internal data representation. As such, the data requirements set by reality are not modeled. Accordingly, they cannot be represented or resolved in a software implementation.

Example 2.2.1 (Erroneous path situation)**.** *The creditworthiness of a customer is checked prior to approving his order. Consequently, this order is accepted and delivered. If that customer goes bankrupt, it is apparent that the order should not have been delivered (in reality). However, no errors are shown in the system. More specifically, the respective process is not required to be modelled in the system.*

As shown in Example 2.2.1, a decision made on certain data may eventually be wrong if that data changes during execution. Such a situation does not lead to data inconsistencies or software errors. It does, however, lead to an erroneous path executed by the system. This is referred to as an *erroneous path situation*.

Accordingly, erroneous path situations may occur during process execution, which may result in unexpected behaviour and undesirable business outcomes. The consequences are often noticed only by end customers (Van Beest et al., 2010b), by erroneous orders or invoices, customer requests that are never handled, etc. The situation where undesirable business outcomes are caused by external data changes is known as *process interference* (Xiao and Urban, 2007; Van Beest et al., 2010a).

The problem of process interference is not centered around the value that is *stored*, but the value that is *used* by the BP and the value that is *correct* in reality. More specifically, the implicit dependency on a value might require a process variable not to change. In many organizations, such a strong semantic overlap exists between the various data repositories of their processes. These process environments lead to those problems that are initially not necessarily experienced inside the organization, as no error messages like a dynamic deadlock detected are signaled. The external part of the disruption, however, has a considerable effect as the data interference induced problem is primarily noticed by the external stakeholders (mostly customers). From a practical point of view, no methods or tools exist that enable the identification of the severity of these problems.

2.3 Business process reconfiguration

Considering the difficulty of design-time verification of business processes, runtime capabilities for adapting to such unforeseen events may provide a more feasible approach. This implies that a currently running instance should be changed on the fly, as the design specifications in the process model are not sufficient to resolve the erroneous situation.

Changeability of business processes is a large research area focusing on providing the capabilities to adapt business processes at designtime or runtime. As a result, flexibility has become very important in information systems and is nowadays an important requirement (Weske et al., 2006). As such, a number of well-known adaptability frameworks have been proposed. The most notable examples are the ADEPT project (Dadam and Reichert, 2009; Göser et al., 2007), and the DECLARE framework (Van Der Aalst et al., 2009).

The ADEPT project is designed to support the synchronization between several running instances of the same process. Any changes made by the user are incorporated into all of the running instances without interrupting their execution (Dadam and Reichert, 2009). An improved version of the framework has been proposed by Göser et al. (2007).

The DECLARE framework utilizes the idea of a declarative process specification (Van Der Aalst et al., 2009) in order to attain flexible process execution. The process defined inside this framework is not a strictly written sequence of actions, but is defined with constraint templates based on temporal logic, which interactively guide the user through the execution of the process.

Weske (2001) provides an approach for enhancing flexibility by dynamic adaptation of running workflow instances. A more detailed overview of various dynamic business process reconfiguration techniques can be found in Rinderle et al. (2004).

Although adaptation of processes to resolve process interference can be considered a very specific form of changeability, existing changeability frameworks are primarily requirements-driven. That is, their adaptation capabilities are specially tailored to facilitate and support new business requirements (and, therefore, improve flexibility), whereas they do not incorporate the mechanisms to adapt the process in order to prevent erroneous business outcomes. Consequently, requirements-driven changeability and adaptability does not solve our research problem, although the ideas may provide valuable contributions to the problem studied in this thesis.

In order to deal with process execution inconsistencies, a number of techniques have been proposed. AGENTWORK is a workflow management system, which supports automated business process adaptations in a comprehensive way. Exceptions and necessary workflow adaptations are specified through a rule-based approach. Using this approach, the system is able to react to process-failures like unavailable resources or data (Müller et al., 2004). Similarly, existing runtime solutions for process interference are based on failing processes as well, e.g. (Garcia-

Molina and Salem, 1987; Xiao and Urban, 2008; Gajewski et al., 2005). That is, only those processes that fail during execution and terminate in an improper way are recovered. In Xiao and Urban (2008), an approach is proposed that deals with recovery of failing processes using dependency tracking based on incremental data changes. A global schedule of these data changes is used to detect data dependencies, in order to determine the impact of process failure and recovery procedures. In practice, however, process interference does not necessarily cause processes to fail. More often, the processes finish regularly without any system errors from an internal perspective, leading however to inconsistent results.

A more elaborate solution for process interference in Service-Oriented Computing is provided by Urban et al. (2011). Predefined (design-time) rules are used to specify the required compensation actions in case of interference. In addition to failing processes, this approach incorporates events like exceptional conditions or unavailable activities. Nevertheless, problems occurring at a regularly executing process due to the use of inaccurate data are not considered.

2.4 Automated runtime reconfiguration

External data changes during execution of a process instance are inevitable and the resulting erroneous path situations are difficult to prevent. Consequently, a runtime solution is required to recognize these situations and act accordingly by reconfiguring the respective process instance in such a way that it provides a desirable process outcome. In the field of Artificial Intelligence (AI), planning techniques have been developed to compose a business process given a set of predefined activities. These *AI planning techniques* may also be used to facilitate reconfiguration of business process.

The advantages of integrating AI planning techniques for several applications in the field of Business Process Management have long been acknowledged. For instance, different planning approaches can assist at the business process definition phase (Rodríguez-Moreno and Kearney, 2002; Rodríguez-Moreno et al., 2007; Madhusudan et al., 2004), while Jarvis et al. (1999) investigate the use of planning in case of domain state changes. In order to facilitate (semi-)automatic adaptation at runtime, AI planning techniques have been used from different viewpoints in the literature. Beckstein and Klausner (1999) discuss the use of an intelligent assistant based on AI planning techniques, which can suggest compensation workflows or the re-execution of activities as a response to execution failures, with the help of meta-level knowledge incorporated in the workflow semantics. The benefits of adding such semantics to BPs have long been acknowledged by the work in the field of Semantic Business Process Modelling, and exploited for a number of different purposes, such as automating process verification (Henneberger et al., 2008), which rely on a description in terms of preconditions and effects, or process model generation (Weber et al., 2010). Preconditions capture the prerequisites of an activity, whereas effects (or postconditions) capture how the activity affects the data.

Ferreira and Ferreira (2006) propose the use of machine learning in order to infer the preconditions and effects of activities, and generate a partially ordered execution plan that complies to these rules. The framework aims at providing a candidate process that is able of achieving some business goals. At execution time, if an activity fails, an alternative candidate plan is provided. Although the objective is different than strictly resolving process interference, a common concern with this framework's approach is the decoupling of the BP-specific constraints from the generic service repository, thus allowing the planner to generate partially ordered plans with a high degree of flexibility.

BP adaptation through planning provides the ability to adapt a running process in case mismatches between the environment and the internal system representation are detected (De Leoni et al., 2007, 2009; Marrella and Mecella, 2011). This work uses several versions of Golog (Levesque et al., 1997), which is based on planning by means of the situation calculus (McCarthy and Hayes, 1969). In Golog the goal to be achieved has to be specified in a procedural way, as a non-deterministic program. This implies that the adaptation process has to be pre-specified in an action-centric way, which requires domain-specific knowledge of the available services and arduous hand-coding by a human expert. One advantage of the approach proposed by Marrella and Mecella (2011) is that it can manage any unforeseen event, by continuously comparing the environment with the expected outcomes according to the BP specification at each step of execution. The approach, however, only provides recovery policies that lead to the expected state as specified in the original process. As a result, it is not able to cover situations as described in Section 1.1, which necessitate the fulfillment of extra requirements or the use of compensation activities.

In order to be able to combine actions in a dynamic way, AI planning methodologies can be adopted for semantic service composition (Sohrabi and McIlraith, 2010; Kaldeli et al., 2011; Au et al., 2005). Many of the approaches proposed for service composition via automated planning, however, require that the set of supported solutions is pre-defined in some form of procedural templates (e.g. (Sohrabi and McIlraith, 2010; Au et al., 2005)). Nonetheless, the domain-independent planner that is presented by Kaldeli et al. (2011) allows the domain designer to use high-level declarative goals by stating *what* properties have to be satisfied, without having to anticipate *how* these can be fulfilled. The planner presented by Kaldeli et al. (2011) can be mapped into a Constraint Satisfaction Problem (CSP), which can in turn be passed to a constraint solver, together with some goal that is expressed in the form of constraints. The computed solution to the CSP (assignment to variables) amounts to an optimal plan (Aiello and Lazovik, 2006) (partially ordered sequence of actions) that satisfies all the constraints imposed by the domain and goal.

CHAPTER 3

Case Study Description

3.1 Case 1: Energy Market

3.1.1 Case description

The energy market (electricity) in the Netherlands is characterized by many complex processes, where often concurrency is involved. Many different stakeholders are involved, including consumers, suppliers, transmission system operators, program managers and measuring companies.

Consumers of electricity conclude a contract for the supply of energy with a supplier. The supplier ensures that during the contract sufficient energy is available to meet the expected demand of the customer. A Transmission System Operator (TSO) is responsible for the electricity network in a certain region. The tasks of a TSO include the construction and maintenance of the energy networks and providing free access to the network for all energy suppliers. A program manager tries to accommodate supply and demand of electricity as well as possible, in order to match the expected consumption of the customers of a particular supplier. The main task of the program manager is to ensure that the electricity supply and demand is balanced at any time of day, in order to avoid underload or overload of the grid. A

Measuring (or Metering) Company (MC) is responsible for reading the electricity meters and sending the readings to the TSOs. Furthermore, the MC is responsible for emplacement and maintenance of the meters. For small consumers, the meter reading is usually recorded once a year by a representative of the MC or by the consumer himself. For large consumers, the readings are supplied monthly.

After liberalization of the energy market, a large number of customers was expected to switch to another supplier. In order to standardize and simplify the information transfer between the various stakeholders, Energy Data Services Netherlands (EDSN) was founded to handle the message exchange of switch requests. Initially established under the name of Energy Clearinghouse (ECH) by Essent and Eneco in 2001, the name was changed to EDSN in 2007. The stakeholders communicate via electronic messages to the system of EDSN, which will forward the messages to the appropriate parties.

EDSN operates as an independent foundation. The participants have no direct control over the foundation and the databases are not accessible other than through electronic messages, in order to ensure protection of sensitive competitive information in the databases. The message exchange never occurs directly between stakeholders, but always via EDSN. Consequently, in case of possible conflicts, it can be retrieved which message is sent by which stakeholder at any time.

Currently, 98% of the electronic messages is sent via EDSN. Through the system of EDSN the entire information transfer needed for the switch-requests and move-requests can be handled. Around 65 000 switch-requests and nearly 60 000 move-requests are handled per month. Furthermore, measuring data (e.g. periodic meter readings) and master data are communicated to the stakeholders. In total, over 100 million messages are processed per year.

3.1.2 Data collection

In order to obtain detailed descriptions of the business processes that are executed in the energy market by the different market parties, documentation of provided by EDSN was used. As all communication between the market parties proceeds via

EDSN, the business processes can be obtained using the message exchange and required data that was available in the documentation. This documentation included sequence diagrams, use case diagrams and class diagrams.

3.1.3 Processes under investigation

In this subsection, the processes used for the energy case will be described subsequently. However, due to a non-disclosure agreement, we will only provide a high-level overview of the processes, in order not to reveal in-depth system details.

Move out

A move out is a rehousing of a customer, where the responsibility and decision-making power of the customer for the connection is ended and transferred. This process is initiated by the customer sending a move out request to his current supplier. The supplier sends the move out request to the TSO. The TSO evaluates the request and changes the connection registry. The supplier then obtains the current meter reading at the customer or through the smart meter. If it concerns a smart meter, the smart meter will be switched off. The supplier validates the raw metering data based on historical metering data. The supplier determines the meter reading and sends it to the TSO. The supplier makes the meter reading available to the measurement registry. Based on this meter reading, the consumption is determined and an invoice is sent to the customer accordingly. A graphical overview of the move out process is shown in Figure 3.1a.

Change of metering responsible

A change of metering responsible concerns the request of a customer to switch the party that is responsible for his connection. This process is initiated by the customer. The metering responsible parties make, prior to the execution of the switch from metering responsibility, agreements on the date of change of control and metering device. The details of a meter change is described below and shown in Figure 3.2. The meter data is exchanged between the responsible parties. The TSO informs the metering responsibles about the switch. The supplier and program responsible are informed through the change in the connections registry. A graphical overview of the change of metering responsible process is shown in Figure 3.1b.

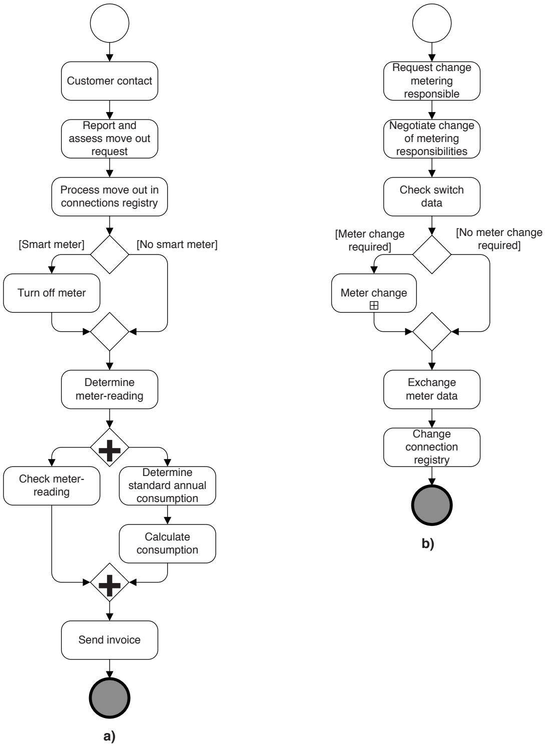

Customer contact
Report and assess move out request
Process move out in connections registry
[Smart meter]
[No smart meter]
Turn off meter
Determine meter-reading
Check meter-reading
Determine standard annual consumption
Calculate consumption
Send invoice
a)
Request change metering responsible
Negotiate change of metering responsibilities
Check switch data
[Meter change required]
[No meter change required]
Meter change
Exchange meter data
Change connection registry
b)

Figure 3.1: Move out (a) and Change of metering responsible (b)

Change of supplier

This process is initiated by the customer, by sending a switch request to the new supplier. The new supplier requests information from the EAN codebook, to obtain address information and connection details. Next, the new supplier checks the contract control protocol to verify whether the customer still has a contract at another supplier. Furthermore, the supplier contacts the measuring registry to obtain metering and consumption information about the customer. The supplier sends a request for change of supplier to the TSO. The TSO evaluates the request and communicates the results to all parties involved. Subsequently, the TSO changes the connection registry and distributes the customer data to the supplier and shipper that are now responsible for the connection. The supplier then obtains the current meter reading at the customer or through the smart meter. If it concerns a smart meter, the smart meter will be switched off. The supplier validates the raw metering data based on historical metering data. The supplier determines the meter reading and sends it to the TSO. The supplier makes the meter reading available to the measurement registry. Based on this meter reading, the consumption is determined and an invoice is sent to the customer accordingly. A graphical overview of the change of supplier process is shown in Figure 3.2a.

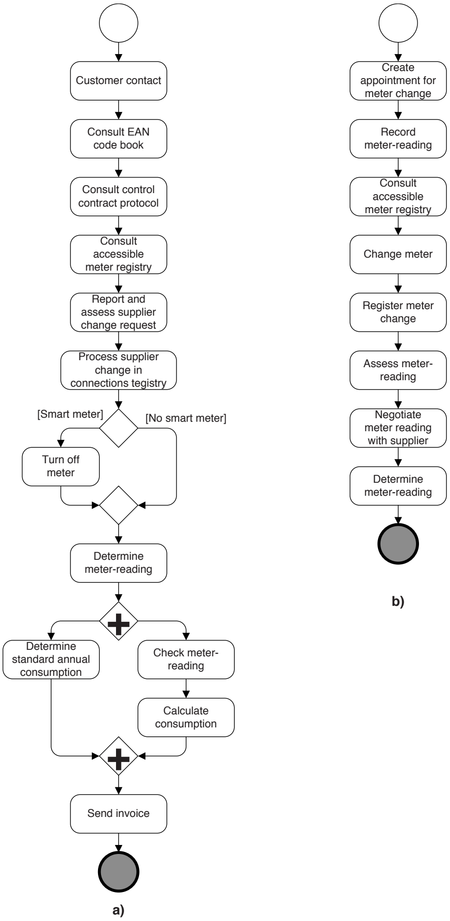

Customer contact
Consult EAN code book
Consult control contract protocol
Consult accessible meter registry
Report and assess supplier change request
Process supplier change in connections tegistry
[Smart meter]
[No smart meter]
Turn off meter
Determine meter-reading
Determine standard annual consumption
Check meter-reading
Calculate consumption
Send invoice
a)
Create appointment for meter change
Record meter-reading
Consult accessible meter registry
Change meter
Register meter change
Assess meter-reading
Negotiate meter reading with supplier
Determine meter-reading
b)

Figure 3.2: Change of supplier (a) and Meter change (b)

Meter change

A meter change concerns the placement, removal, change, failure or calibration of a meter or the replacement of the old meter by a smart meter. This process is initiated by the TSO. The TSO first plans an appointment with the customer and determines the time and date for the meter change. The TSO changes the physical meter and determines the meter reading. The supplier is informed about the meter change and meter reading. The supplier evaluates whether the recorded meter reading is acceptable according to their internal rules. The supplier and TSO negotiate to reach an agreement about the meter reading. Finally, the supplier records the meter reading as negotiated. A graphical overview of the meter change process is shown in Figure 3.2b.

3.2 Case 2: Telecom market

3.2.1 Case description

The telecommunications industry is characterized by many complex processes, involving many different stakeholders, comprising providers, network owners and consumers. A provider of communications provides users of its service the ability to communicate using a computerized device. A provider is responsible for data processing or storing for such a service or for users (consumers) of that service. Consumers conclude a contract with a communication service provider regarding the use of that service. A network owner is responsible for the emplacement and maintenance of the telecommunications network. It can be the same company that also provides the services to the consumers, but in many cases providers use the network owned by a certain network owner.

This case study concerns a company in the telecommunications industry. For confidentiality reasons, the name of the company cannot be revealed and will be referred to as TC. TC is the leading provider of telecommunications and ICT services in the Netherlands, serving customers with both fixed-line and mobile telephony, internet and television. For business customers, TC delivers complete end-to-end telecommunications and ICT solutions. In the Netherlands, TC has well over 6 million fixed-line phone customers. Outside the Netherlands, TC operates under different brandnames. TC has more than 33 million users of their mobile services in the Netherlands, Germany, Belgium, France, and Spain. Worldwide, TC serves more than 40 million customers. In addition to their mobile services, TC provides Internet access to more than 2 million customers. Furthermore, TC offers business network services and data transport in Europe.

Several systems are used to facilitate the customer management business process. The *BPM system* manages the business flows and operational support for the business processes, and interacts with a number of other systems. The most important systems (i.e. the systems that are relevant for the analysis) will be discussed here. The *Front End* allows customers, agents and dealers to manage the customer details, the financial details and the portfolio of the customer. *Infranet* is a back end system, which stores the customers, portfolio definition, and rating and billing. Finally. the *Provisioning Interface* is an application that manages the access rights for all services and customers and is the interface to get and update provisioning information.

3.2.2 Data collection

The process models are obtained from the documentation of the BPM system and from the documentation of the systems it interacts with. This information is interpreted to create a structured overview comprising the different processes, interface data and process interactions. The processes are defined as Sequence Diagrams, including control-flow constraints such as loops and different conditions.

The data used by the different activities are scattered across system specific documentation. Some documents have clear input and output tables for activities, while other documents required specific domain knowledge to extract this information. System specific documents are necessary to extract the information regarding which attributes are used by these different activities.

3.2.3 Processes under investigation

In this subsection, the processes used for the telecom case will be described subsequently. However, due to a non-disclosure agreement, we will only provide a high-level overview of the processes, in order not to reveal in-depth system details.

Buy packages and options

This process is used to buy additional products and services like ADSL, VoIP and mail and to buy additional options on those services. This process can be triggered manually by a customer agent (through the front-end system), as well as by the customer itself using the webpage where products can be ordered.

After the request for additional packages, first all necessary checks for usernames and availability for e.g. broadband are performed. The order can contain one or more packages, with one or more options that need to be activated. If a mailbox already exists, but is inactive, the order must be removed as activation of the existing mailbox is sufficient. For accounts that are activated, the account status must be changed. The customer is informed about the activation of all selected products and services. Next, the billing process is started (outside the scope of the analysis). A graphical overview of the buy packages process is shown in Figure 3.3a.

Upgrade / Downgrade / Switch
Used for the transition of a customer to another package at some specific future date. This process is triggered in the front-end of the system. The new package will be set to active and the old package will be closed at the end of the upgrade, downgrade or switch. First, the customer details will be obtained. Next, all services of the new and old package will be reviewed. All new services are activated. All changed services are modified and all services to be removed from the customer are deleted. The account details are updated and the customer is notified of the successfull transition of the package. Finally, the invoice will be sent. A graphical overview of the upgrade packages process is shown in Figure 3.3b.

Close customer at end of contract terms
Creates the order for the closure of the customers package at the end of the contract term. The process is triggered manually by an agent in the front-end of the system. First, the customer details will be obtained. Next, it will be checked that there is no pending order on the account, as closure is not allowed when there are pending orders. If there are no pending orders, then for all packages the status is changed to ’frozen’ and the package itself is closed. After all packages are frozen, the account is set to frozen. The account will be closed and the customer will be notified. Finally, the billing process is started. A graphical overview of the close customer at end of contract process is shown in Figure 3.4a.

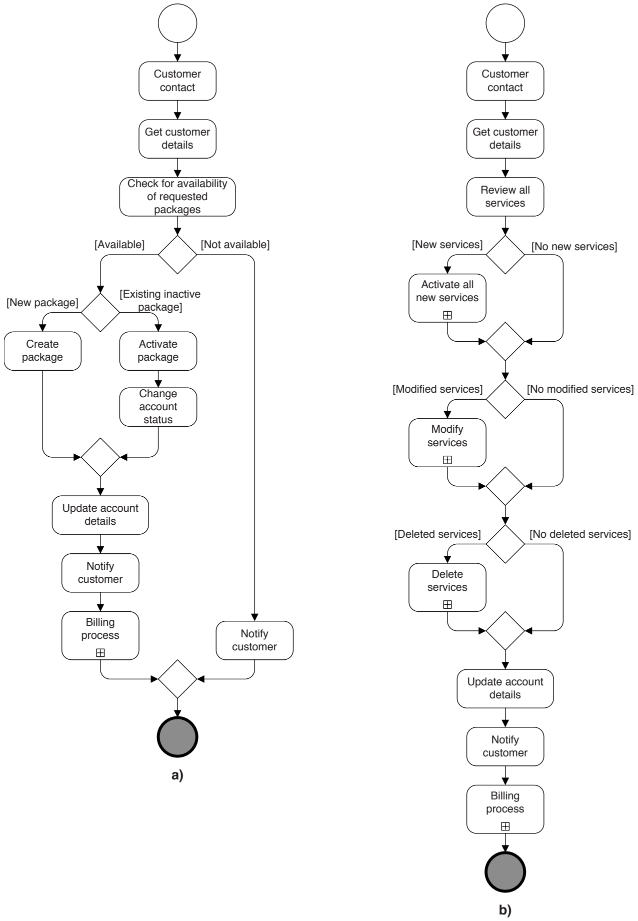


Figure 3.3: Buy packages and options (a) and Upgrade / Downgrade / Switch (b)

Close customer without freezing

This process will close an account of a customer immediately at this point in time. A customer account can be closed with or without freezing. In this thesis, the latter will be analyzed, which implies that the username and mailbox alias corresponding to that customer cannot be recovered. This process is triggered manually by an agent in the front-end of the system, with a request to close the customer. First, the customer details will be obtained. For all packages, the status is changed to 'frozen' and the package itself is closed. Subsequently, all existing sub-accounts are closed. After all packages and sub-accounts are closed, the main account is closed and the customer will be notified. Finally, the billing process is started. A graphical overview of the close customer without freezing process is shown in Figure 3.4b.

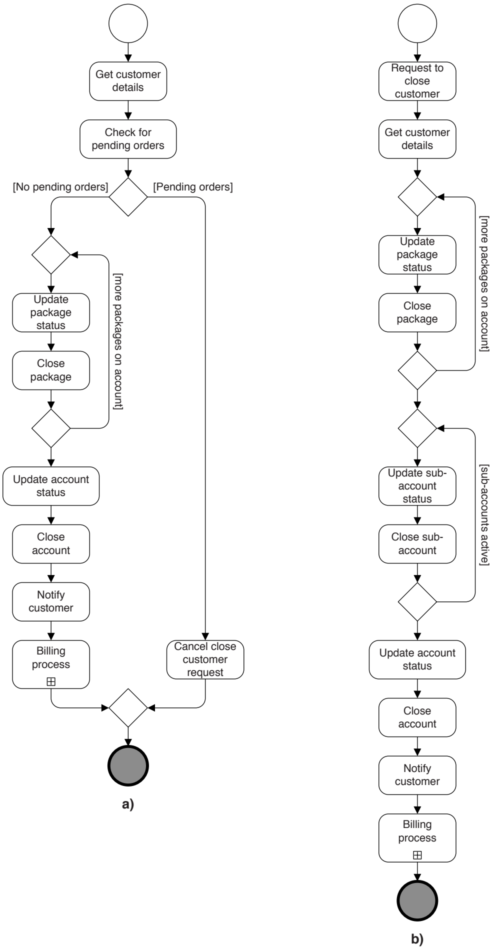


Figure 3.4: Close customer at end of contract terms (a) and Close customer without freezing (b)

Customer move

This process is used to facilitate a move of a customer. The physical provision of the telecom package should be switched to the new location as well. This process is triggered by the customer notifying that he will move at a certain date. The customer details will be updated in the system and a move order is created, as all services and provisions of that customer need to be transferred as well. All services are transferred to the new address and the packages will be updated. A graphical overview of the customer move process is shown in Figure 3.5a.

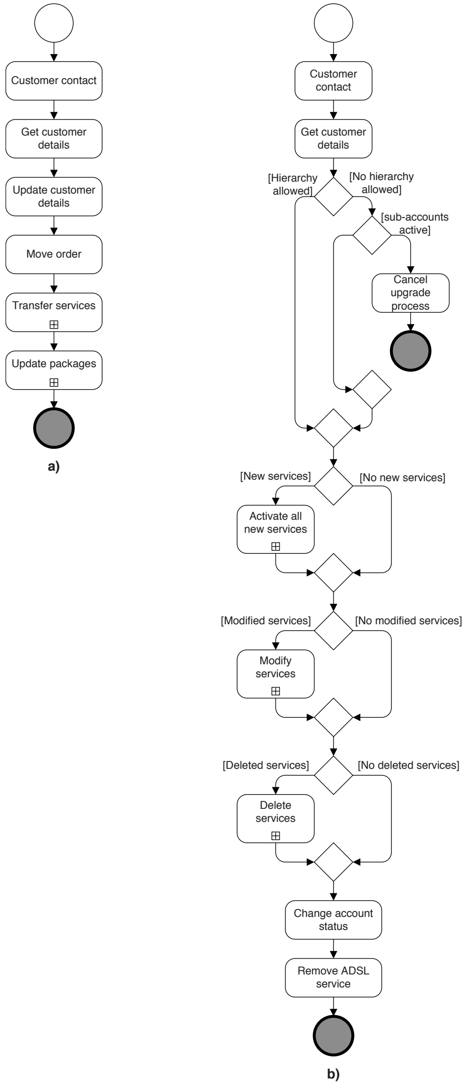


Figure 3.5: Move customer (a) and Upgrade from ADSL to VOIP / IPTV / Broadband (b)

Upgrade from ADSL to VOIP / IPTV / Broadband

This process is used for an upgrade from ADSL to one of the other (broadband) packages. This process can be triggered by a customer agent, or by the customer itself. First, the customer details will be obtained. If the new package does not allow to have subaccounts, it will be checked whether there are non closed subaccounts. If that is the case, the task is set to error and the process stops. Next, all new services are activated. All changed services are modified and all services to be removed from the customer are deleted. Finally, the account status will be changed, setting the new package to active. The ADSL package will be closed. A graphical overview of the upgrade from ADSL process is shown in Figure 3.5b.

3.3 Case 3: Local government

3.3.1 Case description

This case-study concerns a BP with respect to the Dutch Law for Societal Support (known as the WMO law), which was introduced in January 2007. The WMO law replaced the Welfare Law, the Law for Provisions for Disabled (WVG) and parts of the General Law for special diseases (AWBZ). The Welfare Law and the AWBZ were executed by the central Dutch government and the WVG was performed by local municipalities. Currently, the entire WMO law is executed locally at municipalities (Ministry of Health and Sport, 2008).

The WMO law is intended to enable people with a chronic disease or a disability to take part in society and live in their own homes for as long as possible. In order to offer support for such citizens, facilities are provided including domestic care, transportation, a wheelchair or a home modification. The WMO law applies to every municipality in the Netherlands and offers the same service to their citizens. However, the service-level and priorities may differ for each municipality and some differences may exist in the execution of their processes as the responsibility for the execution is defined locally at the municipalities (Ministry of Health and Sport, 2008).

3.3.2 Data collection

The process descriptions available of the WMO processes are very generic, as the details are determined by the municipalities. In addition, documentation at the local municipalities is often incomplete or non-existing. As a result, available process models can only be used as a starting point and details have to be obtained through interviews with process experts at the municipalities. Interviews were conducted at seven different municipalities in the Netherlands, including Delfzijl, Groningen, Haren, Leek, Marum, Winschoten and Winsum.

The employees that were interviewed at the municipalities were either the coordinators of the WMO department or the WMO consultants. With the information gathered from the interviews and the documentation, process models were constructed for the seven municipalities. The processes as executed by the different municipalities differed on some details (i.e. variety of services provided, eligibility criteria, etc.). From these models, a generic process model is constructed that will be used throughout this thesis. The steps taken to generalize the process models are as follows:

1. The original processes are compared to find the common activities that are used to form the basis of the generic model.
2. Activities that are specific to some of the municipalities are modelled as alternate options or left out in case it concerned a situation applicable to a certain municipality only.

Consequently, the obtained process model is not the prescribed model, as it is obtained by talking to the employees that are actually executing that process. For an extensive overview of the methodology for obtaining the generalized process, the reader is referred to Bouma (2010).

3.3.3 Process under investigation

The BP under investigation, referred to as the WMO process, concerns the handling of the requests from citizens at one of the 418 municipalities in the Netherlands. In this section, the WMO process is described as used by one of the municipalities and annotated with the required dependency scopes.

Municipalities are obliged to have a WMO service desk, where the citizens can access all WMO provisions. In some cases, the Intermunicipal Social Service provides the service desk for multiple adjacent municipalities. The WMO process (shown in Figure 3.6) starts with the submission of an application for a provision by a citizen at the local service desk or online. After receiving the application at the municipality office, a home visit is executed by an officer, in order to gather a detailed understanding of the situation and the current living conditions of the citizen. If the home visit is not sufficient to obtain all required information (concerning the citizen's health), a medical advice can be requested from a medical specialist. Based on this information, a decision is made by the municipality to determine whether the citizen is eligible to receive the requested provision or not.

In case of a negative decision (i.e. the application is rejected or the granted provision is less than the citizen requested), the citizen has the possibility for appeal. In case of a legitimate appeal, the provision is either granted, or the process is restarted. In case of a positive decision, the appropriate activities are executed, depending on the requested provision. For domestic help, the citizen has the choice between "Personal Budget" and "Care in Kind". In case of a "Personal Budget", the citizen periodically receives a certain amount of money for the granted provision to pay for workers or supervisors, and decide where the money is spent. In case of "Care In Kind" suppliers who can take care of the provision are contacted. A home modification involves a tender procedure to select a supplier, prior to execution of the actual home modification. A wheelchair is usually provided using a contracted supplier. After acquiring the detailed requirements, the order is sent to the selected supplier, who delivers the provision. After that point, the process is identical for all provisions. The order is sent to the selected supplier, who delivers the provision and sends an invoice to the municipality. Finally, the invoice is checked and paid.

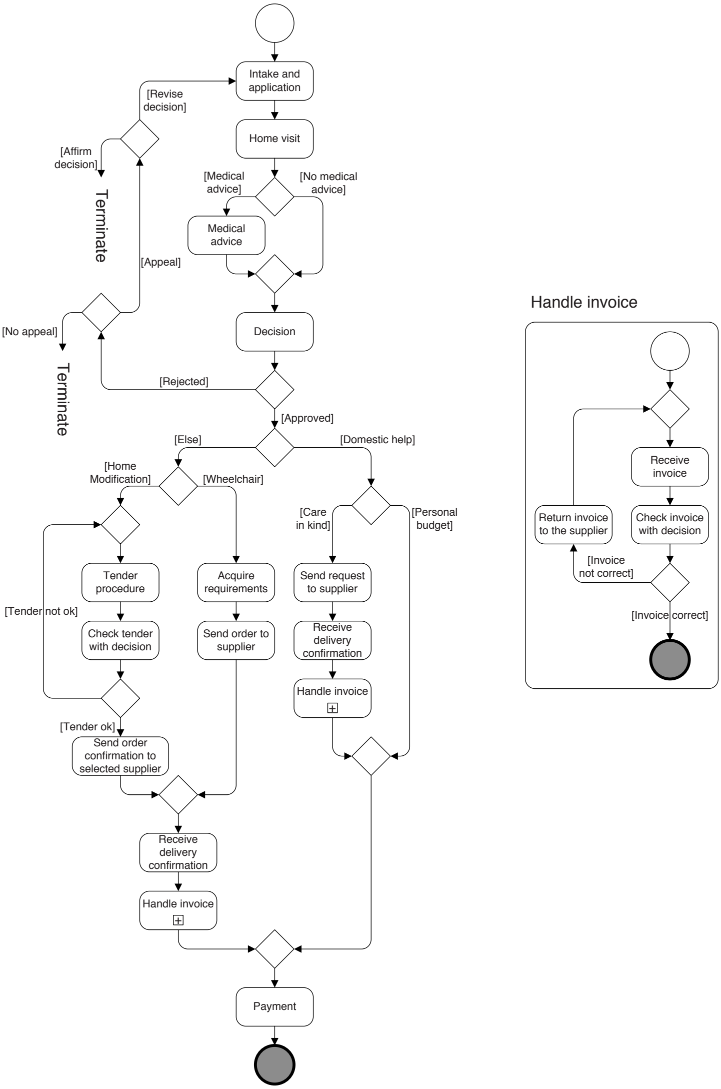


Figure 3.6: WMO process model

CHAPTER 4

Process interference identification

In the previous chapters, it was shown that concurrently executed processes are often assumed to run independently. However, arguably this is often not the case in practice. Existing approaches for interference discovery are not suited for identifying such erroneous situations. That is, the process may be sound when executed in isolation, but may yield in practice undesirable results. This is caused by the semantic overlap between the various data repositories of these processes. To the best of our knowledge, no methods or tools exist that enable the identification of the severity of these problems. Therefore, a methodology is necessary to explicitly define the necessary steps to be performed in order to identify and pinpoint potentially interfering processes.

In this chapter, an investigative method is presented to automatically discover data / process interdependencies between two business processes in order to identify all potential interference situations. The method was applied to two distinct cases, involving the Energy Company (EC) and the Telecom Company (TC) (see Chapter 3). The case studies serve the following purposes: the appropriateness of the method is verified, the utility of the tool is validated and the relevance of the problem is confirmed.

4.1 Interference definition

In this section, process interference will be explained in more detail. In conjunction with a graphical description, process interference will be defined formally. The formal definition is required to specify the exact criteria and properties of process interference for the analysis. The selection of processes for analysis requires the data characteristics and data overlap between two processes. Furthermore, the interference characteristics for erroneous cases (i.e. the order of data changes) are essential to ensure that the identified problems are exclusively on account of process interference.

4.1.1 Graphical example

Let us first recall the graphical description of process interference, as provided in Section 1.1.1. In Figure 4.1, two independent concurrent processes are shown using a common data store. A mutation of a data element by one process, affects the value of the corresponding data element of the other process as well.

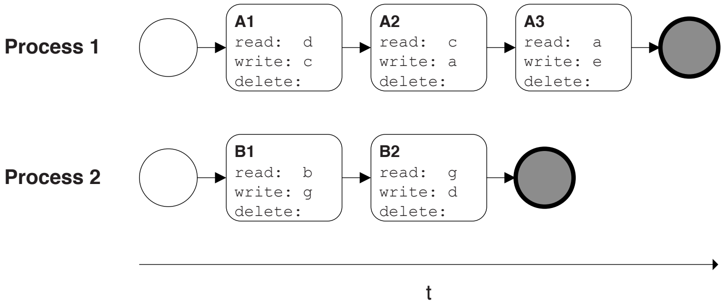


Figure 4.1: Business process with concurrent data change.

Activity A1 reads d and writes the result of a computation based on d to c. Subsequently, activity A2 reads the value of c and writes to a accordingly. Finally, activity A3 reads a and writes to e. Implicitly, the value of e is by transitivity dependent on d.

If a concurrent process changes that value of d (activity B2) *after* it has been read by process 1, potentially e will have the wrong value assigned.

An external data change may also affect the consecutive subprocesses after an evaluation of a condition. In Figure 4.2, for example, the decision made is based on the value of d. That is, the decision activity *uses* d. That specific decision determines whether A1 and A2 are executed or A3 and A4. If d is changed by another process (e.g. process 2) during execution of A2, this may have consequences for the correctness of the activities being executed at that time. That is, as a result of the data change, currently the wrong branch of activities is executed.

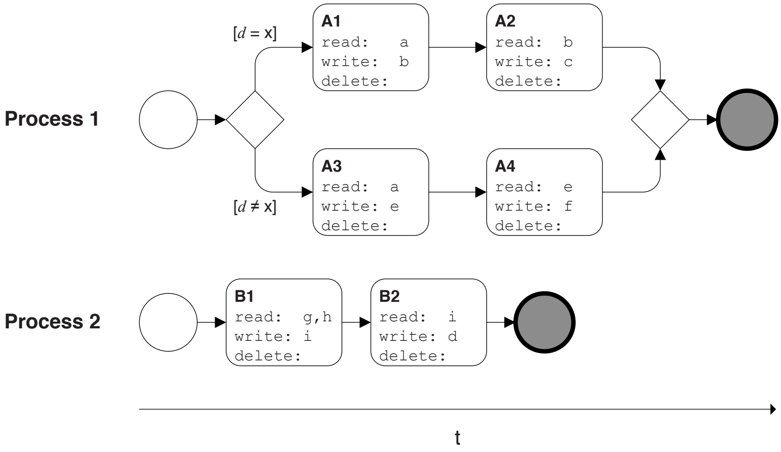


Figure 4.2: Conditional branches with concurrent data change.

In the context of data storage, CREATE, READ, UPDATE and DELETE (CRUD) are the four basic functions (Martin, 1983). A CREATE operation is used for inserting new data. A READ operation is used for retrieving data. An UPDATE operation is used to change data, whereas a DELETE operation is used for removing data. In this context, we will consider the READ operations of an activity to be used for retrieving data, wheras the WRITE operations of an activity include the CREATE, UPDATE and DELETE operations. Furthermore, those operations that *use* the data for writing to another data element will also be considered WRITE operations.

4.1.2 Defining process interference using temporal logic

After providing an informal description of process interference, now a formal definition will be provided in order to be able to pinpoint the exact characteristics of interfering concurrent business processes.

Temporal logic is a formalism that is used for representing propositions qualified in terms of time. Sequences of transitions between states are described, where the future is not yet determined. That is, there is a number of possible paths where one of those paths may be realized. CTL* is a powerful temporal logic that combines branching-time and linear-time operators (Clarke et al., 1986; Emerson and Halpern, 1986).

For the formal definition of process interference, CTL* will be used to specify its temporal characteristics. CTL* is typically defined on a Kripke structure (Clarke et al., 1999):

Definition 4.1.1 (Kripke structure)**.** *Let AP be a set of atomic propositions. A Kripke structure M over AP is a triple $M = (S, R, L)$, where:*

- *S is a finite set of states.*
- *$R \subseteq S \times S$ is a transition relation. For each $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$.*
- *$L : S \rightarrow 2^{AP}$ is a labelling function with the set of atomic propositions that are true in that state.*

Definition 4.1.2 (CTL* syntax)**.** *The language of well-formed CTL* formulas is generated by the following grammar:*

$$\phi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \textit{A}\phi \mid \textit{E}\phi$$
$$\psi ::= \phi \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \textit{X}\phi \mid \textit{F}\phi \mid \textit{G}\phi \mid \phi\textit{U}\phi$$

where:

- $p \in AP$.
- *ϕ is a state formula.*
- *ψ is a path formula.*

A path in M is a sequence of states $\pi = s_0, s_1, \ldots$, such that $(s_i, s_{i+1}) \in R$ for every $i \geqslant 0$. The suffix of π starting at s_i is denoted by π^i.

Definition 4.1.3 (CTL* validity)**.** *Given a state formula ϕ and a path formula ψ, ϕ holds at state s ($s \models \phi$) and ψ holds at path π ($\pi \models \psi$), as can be inductively defined:*

$$
\begin{array}{lcl}
s \models p & \Leftrightarrow & p \in L(s) \\
s \models \neg\phi & \Leftrightarrow & s \not\models \phi \\
s \models \phi_1 \vee \phi_2 & \Leftrightarrow & s \models \phi_1 \quad \textit{or} \quad s \models \phi_2 \\
s \models \phi_1 \wedge \phi_2 & \Leftrightarrow & s \models \phi_1 \quad \textit{and} \quad s \models \phi_2 \\
s \models \textit{E}\,\psi & \Leftrightarrow & \textit{there is a path}\ \pi\ \textit{from}\ s\ \textit{such that}\ \pi \models \psi \\
s \models \textit{A}\,\psi & \Leftrightarrow & \textit{for every path}\ \pi\ \textit{from}\ s,\ \pi \models \psi \\
\pi \models \phi & \Leftrightarrow & s\ \textit{is the first state of}\ \pi\ \textit{and}\ s \models \phi \\
\pi \models \neg\psi & \Leftrightarrow & \pi \not\models \psi \\
\pi \models \psi_1 \vee \psi_2 & \Leftrightarrow & \pi \models \psi_1 \quad \textit{or} \quad \pi \models \psi_2 \\
\pi \models \psi_1 \wedge \psi_2 & \Leftrightarrow & \pi \models \psi_1 \quad \textit{and} \quad \pi \models \psi_2 \\
\pi \models \textit{X}\,\psi & \Leftrightarrow & \pi^1 \models \psi \\
\pi \models \textit{F}\,\psi & \Leftrightarrow & \textit{there exists a}\ k \geqslant 0\ \textit{such that}\ \pi^k \models \psi \\
\pi \models \textit{G}\,\psi & \Leftrightarrow & \textit{for all}\ k \geqslant 0,\ \pi^k \models \psi \\
\pi \models \psi_1\ \textit{U}\,\psi_2 & \Leftrightarrow & \textit{there exists a}\ k \geqslant 0\ \textit{such that}\ \pi^k \models \psi_2\ \textit{and} \\
 & & \textit{for all}\ 0 \leqslant j < k,\ \pi^j \models \psi_1
\end{array}
$$

CTL* formulas consist of *path quantifiers* and *temporal operators*. Path quantifiers specify whether some or all paths should have a certain property starting at the currens state. The following path quantifiers can be distinguished:

- Aϕ All: ϕ has to hold on all paths starting from the current state.
- Eϕ Exists: there exists at least one path starting from the current state where ϕ holds.

Temporal operators describe properties of a path through the computation tree. Five temporal operators can be distinguished:

- Xϕ Next: ϕ has to hold at the next state.
- Gϕ Globally: ϕ has to hold on the entire subsequent path.
- Fϕ Finally: ϕ eventually has to hold.
- ϕUψ Until: ϕ has to hold at least until ψ holds.

After introducing CTL*, a translation of a process to Kripke structures is necessary to facilitate the definition of the temporal properties of process interference. Let us first define a Kripke structure M for a process P. The variables used by the process P are defined by the finite set of data elements D. In process 1 in Figure 4.1, $D = \{a, c, d, e\}$. The set of states S of a Kripke must include information about the *operation* that is performed on the data when a transition is executed, as that is required to capture the process execution trace in case of process interference. As such, a state $s \in S$ is defined as $\{\mathbf{r}(d) \mid d \in D\} \cup \{\mathbf{w}(d) \mid d \in D\}$, where $\mathbf{r}(d)$ is a READ operation and $\mathbf{w}(d)$ is a WRITE operation. Atomic propositions are all READ and WRITE operations. An activity $a \in P$ maps to a transition relation as follows: a transition concerns the move from one state to another by capturing all operations of a, which are stored in L. There exists one transition relation for each activity $a \in P$. Consequently, $(s_i, s_j) \in R \Leftrightarrow \exists a \in P : s_i \rightarrow s_j$. Each activity can comprise both READ and WRITE operations. Correspondingly, L is defined for each state by operations of an activity that brings the process to that state. In Figure 4.1, activity A1 performs a READ operation on d and a WRITE operation on c. Subsequently, the transition that maps to A1 comprises the READ and WRITE operations, which result in the atomic propositions represented by s_1. This is represented graphically in Figure 4.3.

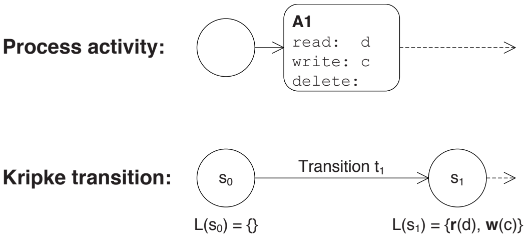


Figure 4.3: Creating a transition relation for an activity.

However, the modeling of a process interference situation (such as shown in e.g. Figure 4.2) requires two processes to be represented in one Kripke structure. In order to represent these two processes in a Kripke structure, an asynchronous composition of Kripke structures is required, to formulate a joint Kripke structure that incorporates both processes. This is shown graphically in Figure 4.4.

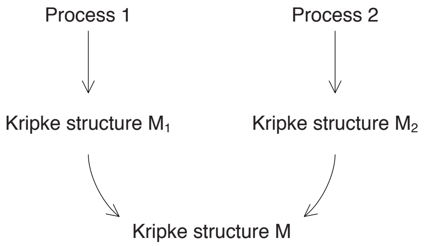


Figure 4.4: Creating a composition of Kripke structures based on two processes.

Note that this translation process follows ”classical” translation algorithms, such as Clarke et al. (1999). However, we focus primarily on the Kripke representation. Although there are many ways in literature to represent a process as a Kripke structure (see e.g. Clarke et al. (1999), Trčka et al. (2009) and Bucur and Kwiatkowska (2011)), being thorough on this topic is beyond our purpose. Consequently, we are rather agnostic, and therefore informal, in the conversion presented here.

Definition 4.1.4. (Interleaved asynchronous composition of Kripke structures). *Given a Kripke structure $M_1 = (S_1,\ R_1,\ L_1)$ over AP_1 and a Kripke structure $M_2 = (S_2,\ R_2,\ L_2)$ over AP_2, the interleaved asynchronous composition $M = (S,\ R,\ L)$ over AP can be defined as:*

- $D = D_1 \cup D_2$
- $AP = AP_1 \cup AP_2$
- $S = S_1 \times S_2$
- $s_{(0)} = (s_{1(0)}, s_{2(0)})$
- $((s_1^{'}, s_2^{'}), (s_1^{''}, s_2^{''})) \in R \Leftrightarrow$
  $((s_1^{'}, s_1^{''}) \in R_1 \wedge s_2^{'} = s_2^{''}) \vee ((s_2^{'}, s_2^{''}) \in R_2 \wedge s_1^{'} = s_1^{''})$
- $L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$

The composition set of data elements D is the union of D_1 and D_2. The composition set of atomic propositions AP is the union of AP_1 and AP_2. The domain of a data element $dom(d \in D) = \{r, w\}$. As a result, $S = dom^{|D|}$. Therefore, the composition set of states S is the cartesian product of S_1 and S_2. Consequently, the initial state s_0 is the set of initial states of both processes $(s_{1(0)}, s_{2(0)})$. A transition from a state $(s_1^{'} \in S_1, s_2^{'} \in S_2)$ to a state $(s_1^{''} \in S_1, s_2^{''} \in S_2)$ exists if and only if the transition concerns an advance of one process. That is, the two processes cannot advance their states at the same time[1]. Finally, the labeling function L is the union of the labeling functions L_1 and L_2 of both processes.

Using CTL* and the interleaved asynchronous Kripke representation, the temporal characteristics of process interference between two concurrently executed processes can now be defined formally. The READ and WRITE operations of $process_i$ on data element $d \in D$ will be denoted by $\mathbf{r}_i(d)$ and $\mathbf{w}_i(d)$ respectively.

Definition 4.1.5 (Process Interference)**.** *If $process_1$ and $process_2$ are executed concurrently, process interference concerns the situation where:*
$EF[\mathbf{r}_1(d) \wedge EF[\mathbf{w}_2(d) \wedge (\neg \mathbf{r}_1(d)\ U\ \mathbf{w}_1(d))]$*, with:*

- $\mathbf{r}_1$: READ *operation of* $process_1$
- $\mathbf{w}_1$: WRITE *operation of* $process_1$
- $\mathbf{r}_2$: READ *operation of* $process_2$
- $\mathbf{w}_2$: WRITE *operation of* $process_2$
- d : *data element used in* $process_1$ *and* $process_2$
- $process_1 \neq process_2$

Two processes interfere if there is a path where d is read by process 1 and d is written by process 2 on some continuing path (i.e. $EF[\mathbf{r}_1(d) \wedge EF[\mathbf{w}_2(d) \ldots]$) such that on some following path d is not read again by process 1 until process 1 performs a WRITE operation on d (i.e. $\ldots \wedge (\neg \mathbf{r}_1(d)\ U\ \mathbf{w}_1(d))]$).

The different data anti-patterns defined by Trčka et al. (2009) include a distinction

[1] Please note that it is assumed that two transitions from two processes cannot be executed at the same time. A further explanation about this assumption will be provided in Section 4.5.1.

between weak and strong variants of interference. In our case, this distinction is not applicable, as we anyway only identify *potential* problems: a “strongly interferable” process implies interference in *all* possible executions. That is, *any* (partially) concurrent execution of some other process would *definitely* cause interference, which should be considered a design error. For instance, such a situation does not occur in a serialized execution of the processes. Weak interference is, therefore, the only one. The interference definition provided in this thesis covers the situation where a situation may happen (i.e. weak interference), which is a sufficient incentive to take action accordingly (as shown in detail in Chapter 5 and 6).

Note that the temporal characteristics referred to in the definition do not necessarily imply erroneous path situations, or erroneous outcomes in general. Process interference does not necessarily result in erroneous outcomes. That is, if a situation complies with the temporal characteristics of process interference, this does not necessarily result in erroneous outcomes. However, if such erroneous path situations occur due to process interference, the situation does comply to these characteristics.

4.2 Method description

Process interference is defined as a model-checking problem. Using this temporal definition of process interference, the EC and TC processes can be analyzed. However, the process interference definition only refers to a data change, but not whether that particular change is actually disruptive for that process. Therefore, the formal description in Definition 4.1.5 cannot be used to predict erroneous path situations, as it does also capture false positives (situations where two processes interfere, but they do not result in erroneous outcomes).

In order to identify the situations that provide erroneous outcomes, an overview is required of all possible outcomes for selected pairs of processes. Therefore, a methodology will be presented using an exhaustive search to obtain such an overview. That is, instead of executing the model-checking algorithm, a simulation will be executed. In Figure 4.5, a schematic overview of the methodology is provided. The following steps should be subsequently executed:

- Gather initial process documentation.
  First of all, information about the available business processes needs to be obtained, including all activities along with their input and output fields.

- Select business processes.
  Interference is expected for two processes where at least 1 process has a READ operation on datafield d and the other process has a WRITE or DELETE operation on d. Consequently, a selection should be made of business processes that have data-overlap and are, therefore, potentially vulnerable for process interference.

- Perform combinatorial analysis.
  For each pair of selected processes, a combinatorial analysis will be performed of each potential execution path to identify the difference between the desired outcome (i.e. when both processes are executed in isolation or in sequence) and the outcome with parallel execution. All possible combinations of parallel execution are simulated step by step, where every READ operation is proprietary to the process and every WRITE operation affects all following reads of both processes. The amount of candidate solutions can be reduced to a manageable size using a set of problem-specific heuristics.

- Analyze erroneous outcomes.
  Finally, the identified erroneous execution orders are verified against the business process specifications.

In the next sections, these steps will be explained in detail.

4.3 Initial data gathering, cleaning and structuring

The information about the business process used for the analysis can originate from documentation or can be gathered by process mining on existing systems (Van Der Aalst et al., 2003c). The latter requires a number of additional steps prior to the actual data gathering (as described in e.g. (Măruşter and Van Beest,

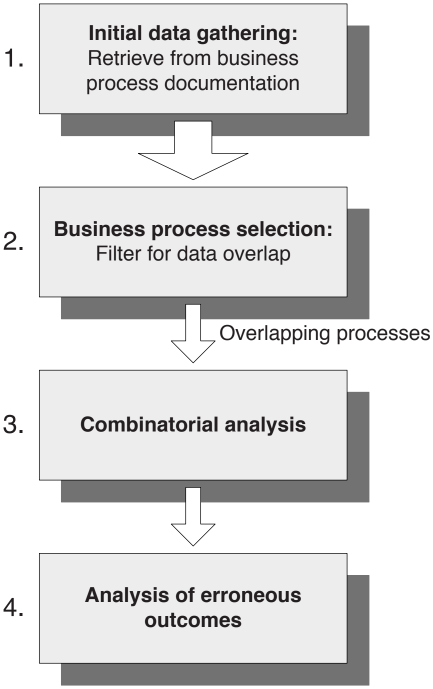

1.
Initial data gathering:
Retrieve from business process documentation
2.
Business process selection:
Filter for data overlap
Overlapping processes
3.
Combinatorial analysis
4.
Analysis of erroneous outcomes

Figure 4.5: Analysis methodology.

2009)). In the case studies described in this chapter, documentation was used comprising detailed system information, where all services and interface details are described in detail. The processes described in Chapter 3 show the high-level business process, whereas the documentation provides an indepth description of the underlying system. The majority of the activities as specified in the system are executable. Although systems may be used in different ways than envisioned, the results obtained by this analysis are still usable. That is, if interference related problems can be identified in the processes specified, they will very likely exist in reality as well.

Apart from the initial data gathering, a number of additional data cleaning and structuring steps are required, in order to make the process information suitable for analysis. First of all, existing process documentation may contain inconsistencies concerning naming policies of activities and data. For example, a telephone number represented by $telephonenr$, may also be referred to as $telnr$ or tel_no. Although these refer semantically to the same concept in reality (i.e. telephone number), they may be represented differently in the documentation. Such inconsistencies in naming policies either concern mistakes in naming conventions (and occur, therefore, in the documentation only), or concern the representation of a field in multiple semantically overlapping data repositories (and are, therefore, a correct representation of reality).

In both cases, however, these fields should be marked as being a representation of the same field in reality (and as a result, being treated as such in the analysis), in order to be able to identify the consequences of a change in one of these fields. Therefore, these synonyms need to be found and provided with a univocal name.

Next, for each process, the activities need to be distinguished in order to identify potential data overlap between the processes. As explained in Chapter 2, every activity in a business process requires data for its execution and some activities may affect the data.

As the formal definition of process interference indicates, it is necessary to distinguish for each datafield used by an activity whether it is read or written. That is, the READ/WRITE distinction is to be made at the datafield level, because an activity may contain both readfields and writefields. A readfield is a field that is read (i.e. *not changed*) by the activity. A writefield is a field that is *changed* by the activity.

For example, a WRITE activity contains both readfields and writefields. That is, a WRITE activity updates a certain data object using the input of the activity. However, a WRITE activity may in addition return a result (the output, which is a confirmation or an error message) to be read by the service or stakeholder that requested the WRITE activity. Consequently, there exist returned results in the form of a datafield which is used by the WRITE activity, but which is not changed. Accordingly, not all datafields that are linked to a WRITE activity as described in the documentation are necessarily writefields.

Input- and outputfields are used differently by an activity depending on the nature of the activity itself (READ or WRITE). For example, a READ activity may require a key as input. This key may be used to compute or obtain the values to be read by that activity (which is de resulting output). A WRITE activity makes a change to the data, by means of a CREATE, UPDATE or DELETE operation. The datafields to be changed are used as input for the WRITE activity, whereas the returned result will contain a message confirming a successful execution of that WRITE activity.

Consequently, each activity needs to be categorized as either a READ activity or a WRITE activity. Although a data object does (obviously) not exist before the CREATE operation, it might lead to redundant data if another process already has created that data. In contrast with an UPDATE operation, a DELETE operation applies to an entire object, rather than a single field. Deletion of a record will be based on a key, deleting the associated object. However, regardless whether it concerns an UPDATE or DELETE, some data is changed. Therefore, all these changes to data (i.e. CREATE, UPDATE and DELETE) will be referred to as a WRITE activity for the remainder of this paper. Once all activities have been categorized, the data used in each of these activities should be marked as either a readfield or a writefield.

4.3.1 The case of the Energy Company

For the EC case, each activity consists of both a request and a response service. For instance, the *Call for Move Out* activity has a *Call for Move Out Request* service and a *Call for Move Out Response* service. In order to execute the *Call for Move Out* activity, the energy provider sends a *Call for Move Out Request* to the Connections Registry. The Connections Registry sends a *Call for Move Out Response* back to the energy provider. Consequently, a WRITE activity has both a write request service and a write response service. Similarly, a READ activity has both a read request service and a read response service. The fields used for each service can be extracted from the data model that is part of the documentation. Table 4.1 shows how the fields for each service type are to be used in the analysis. The EC documentation marks whether the activities are READ or WRITE activities.

The EC documentation distinguishes READ services from WRITE services. However, the EC documentation does not distinguish input and output fields, but distin-

Activity type:	Service type:	Fields used as:	Example:
Read	Read request	Readfields	Identifier (PK) etc.
	Read response	Readfields	Data read
Write	Write request	Writefields	Data to be written
	Write response	Readfields	Return message (ok, or error)

Table 4.1: Overview of read and write indicators of the EC case

guishes input and output services instead. That is, the fields associated with the request service correspond to the inputs to an activity, whereas the fields associated with the response service correspond to the outputs of the activity. Every service request is initiated by an actor (e.g. customer, supplier, grid operator etc.) and handled by another actor in the process, who returns the service response.

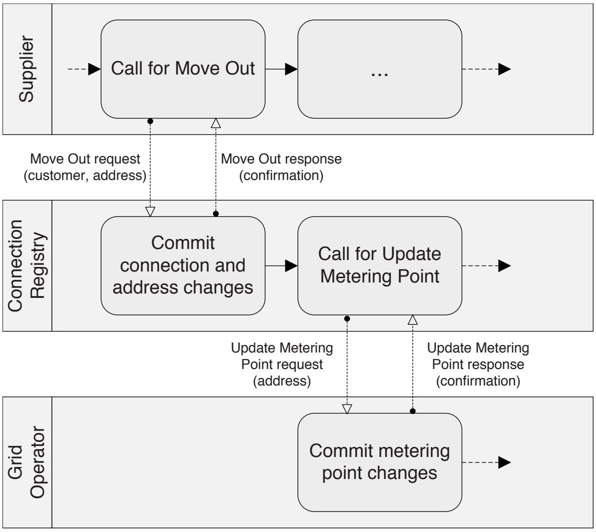

Supplier
Call for Move Out
...
Move Out request (customer, address)
Move Out response (confirmation)
Connection Registry
Commit connection and address changes
Call for Update Metering Point
Update Metering Point request (address)
Update Metering Point response (confirmation)
Grid Operator
Commit metering point changes

Figure 4.6: Simplified example of a part of an EC process (BPMN).

In order to illustrate this distinction between readfields and writefields, a simplified example of a part of an EC process is provided in BPMN notation in Figure 4.6. Both *Call for Move Out* and *Call for Update Metering Point* are WRITE activities. The write request fields *customer* and *address* are writefields, wheras the *confirmation* of both services is a readfield. The corresponding representation in a sequence diagram is provided in Figure 4.7.

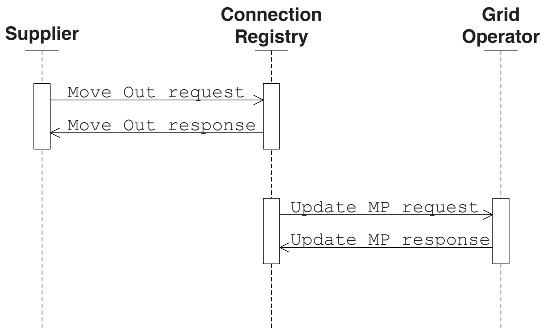


Figure 4.7: Simplified example of a part of an EC process (Sequence Diagram).

4.3.2 The case of the Telecom Company

The business processes of TC are described by a set of sequence diagrams, distributed over several documents. The interface information of all activities are available in separate documents. For each READ and WRITE activity, the input and output fields can be identified. In Table 4.2, an overview is provided of the read and write fields in the TC case.

Activity type:	Fields:	Fields used as:	Example:
Read	Input	Readfields	Identifier (PK) etc.
	Output	Readfields	Data read
Write	Input	Writefields	Data to be written
	Output	Readfields	Return message (ok, or error)

Table 4.2: Overview of read and write indicators of the TC case

In the TC documentation, for each activity, the input- and outputfields can be distinguished. Outputfields of a WRITE activity are the returned result after writing the inputfields (i.e. a message with a confirmation of a successful WRITE operation). Therefore, inputfields of a WRITE activity are treated as writefields, as the value of these fields is *changed*. Similarly, outputfields of a WRITE activity concern a return message and *do not* concern changes in the database. That is, the return message is not written to the database. The return messages can, therefore, be regarded as readfields. The inputfields of a READ activity are used to determine which data to retrieve (i.e. an input parameter, primary key etc.). The outputfields of a READ activity, however, are a result as well. As no data is changed in either the inputfields or the outputfields, all fields are regarded as readfields. Similar to the EC case, every service is initiated by an actor and received by another actor in the process.

4.4 Selection of business processes for analysis

The fields used by the activities of both processes are compared, to identify all potential data overlap between two processes. If data is read only by both processes, no problems can occur as the data is not changed (note that Definition 4.1.5 requires d to be written by at least one of the two concurrent processes). Consequently, overlap in data use is considered potentially harmful if one of the processes (or both) is changing certain data that is also required by the other process. That is, overlap is potentially harmful if the following condition holds:

$$\begin{aligned}\exists d :\ & (r_1(d) \in process_1 \wedge w_2(d) \in process_2) \vee \\ & (w_1(d) \in process_1 \wedge r_2(d) \in process_2) \vee \\ & (w_1(d) \in process_1 \wedge w_2(d) \in process_2)\end{aligned}$$

where:

– r_1 : some READ operation of $process_1$

– w_1: some WRITE operation of $process_1$

– r_2 : some READ operation of $process_2$

– w_2: some WRITE operation of $process_2$

– d : data element used in $process_1$ and $process_2$

Please note that the cases covered by Definition 4.1.5 is a subset of this set of criteria. Instead of checking the formula provided by Definition 4.1.5, we use the set of more relaxed criteria defined above, as it is unfeasible to check for the condition of Definition 4.1.5 for all processes. Therefore, these criteria may potentially result in more process combinations, but these will be eliminated in the analysis (Section 4.7).

Subsequently, overlapping fields need to be filtered on importance for the analysis. That is, each field is filtered based on the severity of the business implications in case these fields are inconsistent. This rating will allow to create a layered representation of the analysis, including only those fields that are important for the analysis, like for example address information and connection information. On the other hand, academic title and salutation are not considered to be important datafields. This filtering for importance is a manual operation and is a context-dependent decision, which is the responsibility of the process analyst.

For the selection of business processes for analysis, pairs of processes were verified on the existence of a datafield d according to the condition stated above. For the EC case, the following processes were selected:

- *Move Out*
  This process concerns the rehousing of a customer, where the responsibility and decision-making power of the customer for the connection is ended and transferred.

- *Meter Change*
  This process concerns the placement, removal, change, failure or calibration of a meter or the replacement of the old meter by a smart meter.

- *Change of Supplier*
  This process concerns the move from one customer to another energy supplier.

- *Change of Metering Responsible*
  This process concerns the request of a customer to switch the Metering Company that is responsible for his connection.

The process pairs formed for analysis of the EC case are shown in Table 4.3 below.

Process 1	vs.	**Process 2**
Change of Supplier		Move Out
Change of Supplier		Meter Change
Change of Metering Responsible		Move Out

Table 4.3: Overview of selected processes for Energy company

For the TC case, six distinct processes were selected:

- *Buy Packages and Options*
  This process is used for handling customer purchases of additional packages like ADSL, VoIP and mail and for additional options on packages.

- *Close Customer Without Freezing*
  This process will close an account of a customer immediately at this point in time, where the username and mailbox alias corresponding to that customer cannot be recovered.

- *Customer Move*
  This process is used to facilitate a move of a customer. The physical provision of the telecom package should be switched to the new location as well.

- *Close Customer at End of Contract Terms*
  This process concerns the closure of the customer at some future date.

- *Upgrade/Downgrade/Switch*
  This process concerns the transition of a customer to another package at some specific future date.

- *Upgrade from ADSL to VOIP/IPTV/Broadband*
  This process is used for an upgrade from ADSL to one of the other (broadband) packages.

The process pairs formed for analysis of the TC case are shown in Table 4.4 below.

Process 1	vs.	**Process 2**
Buy Packages and Options		Close Customer Without Freezing
Customer Move		Close Customer at End of Contract Terms
Upgrade/Downgrade/Switch		Upgrade from ADSL to VOIP/IPTV/Broadband

Table 4.4: Overview of selected processes for Telecom company

4.5 Finding erroneous outcomes through data flow simulation

When two or more independent processes execute concurrently, their activities execute in an *interleaved* fashion. That is, activities from one process may execute in between two activities from another process (Bernstein et al., 1987). If the interleaved execution of two processes produces the same business outcome as the sequential execution of the same processes, these processes are called serializable (Bernstein et al., 1987). However, interleaved execution of activities that use the same data potentially results in process interference (Definition 4.1.5), as the interleaved execution may provide *different* business outcomes than the sequential execution. Consequently, such interfering processes are not serializable. In this section, we will test the serializability of the selected process pairs and identify the potential erroneous results accordingly.

4.5.1 Execution serialization

Due to the request-response character of the activities (as they are supported by web services), the activities (or services) of the processes under investigation are atomic. That is, a service request is either sent or not sent and a service response is received or not received. The execution time of activities (i.e. the time between the service request and service response) is much larger than the execution times of the individual service request and response, which equal a few microseconds. As a result of this negligible small execution time, two services as part of two different processes are assumed not to occur at exactly the same (discrete) time and, therefore, do not overlap during execution. Consequently, the activity order of two processes under investigation can be considered sequential. For example, in a situation with two processes with two activities each (A and B for process 1, P and Q for process 2), this would lead to 6 possible activity execution orders, as shown in Figure 4.8.

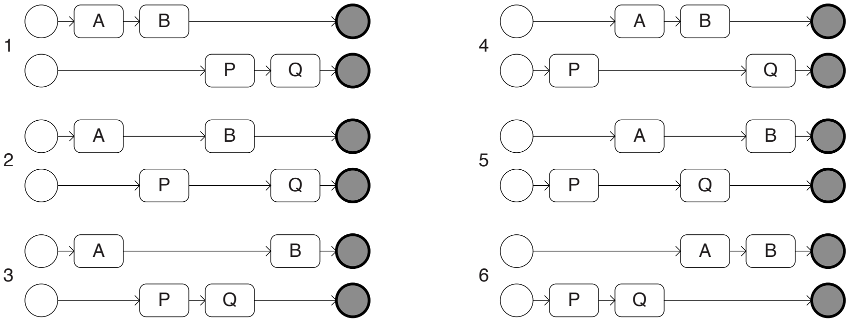


Figure 4.8: Example of execution possibilities for two processes.

4.5.2 Data flow simulation

Process interference can be identified by simulating different execution combinations (such as shown in Figure 4.8) and analyzing the data values (such as shown in Table 8.3) of important fields afterwards. For each combination, the data-flow through the process will be simulated. For all stakeholders / actors in the process the value of all (important) overlapping fields will be stored and monitored during the simulation. Prior to simulation, the desired output of both processes will be determined by executing both processes sequentially. That is, first finish executing $process_1$ prior to executing $process_2$ to eliminate interference.

Next, possible execution orders of both processes are simulated. That is, the output of all possible combinations will be compared with the desired output. If the output for some combination differs from the desired output, this process is potentially vulnerable to process interference for that particular execution order.

4.5.3 Tracking data values during the simulation

In order to illustrate the interactions between stakeholders through activities, an example of service requests between stakeholders is depicted in Figure 4.9. S1, S2 and S3 are stakeholders in the process and activities A, B and C are each supported by two services: a request and a response service. A is a READ activity, where S1 reads the value of d from S2. B is a WRITE activity, where S1 submits a new value of d to S3. C is a WRITE activity, where S2 submits a new value of d to

S3. Each activity has two stakeholders, a source (i.e. the one that sends a request) and a target (i.e. the one that receives the request).

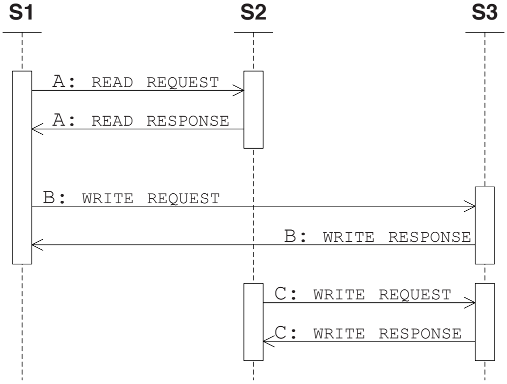


Figure 4.9: Sequence Diagram showing READ and WRITE services between stakeholders.

During the simulation, for each (important) overlapping field the value is stored as known by every stakeholder in the process. That is, for each stakeholder a separate list is kept of the field values as known to that stakeholder. The initial values are marked, so that the origin of the values can be traced during the simulation. As a result, for each field it can be traced which stakeholder received the update of a field and which stakeholders hold the incorrect value. For example, if stakholder A receives an address change, at the end of the simulation the value of the address known to stakeholder B should originate from stakeholder A (who received the actual change to the address) and hold the last value assigned to stakeholder A. If the value of stakeholder B does not originate from stakeholder A or does not contain the last value assigned to stakeholder A, apparently the change to the address as intended was not effectuated to stakeholder B. Consequently, stakeholder B holds the wrong value of the address.

In addition, all WRITE operations are recorded, so that for each field it can be determined whether the values originate from the same WRITE operation or not. For each WRITE operation the responsible process is stored, in order to be able to obtain a WRITE sequence representing the order of write operations to a datafield.

4.6 Tool support

4.6.1 Combinatorial complexity

Using two processes with 4 activities in total yields 6 distinct execution orders (see Figure 4.8). For larger processes, however, the amount of execution orders to be simulated increases rapidly. If p represents the number of activities in $process_1$ and q represents the number of activities in $process_2$, then the amount of distinct execution orders (permutations) is denoted by the following multinomial coefficient (Skiena, 1990, p. 12):

$$\#Distinct\ execution\ orders = \frac{(p+q)!}{p!\,q!}$$

As a result of the combinatorial complexity, the analysis of larger processes will quickly result in an unfeasible amount of combinations. For example, an analysis of two regular processes with 30 activities each would result in more than $1 \cdot 10^{17}$ combinations.

In order to reduce the amount of possible combinations, only those activities should be included in the analysis, whose execution is essential for the final value of a particular datafield and can, therefore, potentially be responsible for a bad outcome. Furthermore, the set of overlapping datafields can be reduced to a smaller subset, by only considering those datafields that are essential. Datafields are considered essential depending on the severity of the business implications in case these fields are inconsistent. For the EC case, these fields concern for example customer data (Address, Financial data), meter data, and connection data. In addition, the analysis can be started from the first *essential* WRITE activity. That is, the first activity that effectively assigns a new value to d is not necessarily the first activity. However, it can be considered the first activity in the analysis, as the preceding activities do not have an effect on d and have no influence on the result. Consequently, the additional measures to reduce the amount of activities in the analysis can be summarized as follows:

1. Reduce the set of overlapping datafields to a smaller subset datafields that are considered essential.

2. Incorporate only those activities which use overlapping data.

3. Start from the first essential WRITE activity. I.e. exclude all activities before that WRITE activity.

However, an analysis of two processes with 15 relevant activities each, would still result in 155 117 520 combinations. It is evident that it is unfeasible to perform such an analysis manually.

4.6.2 The analysis tool

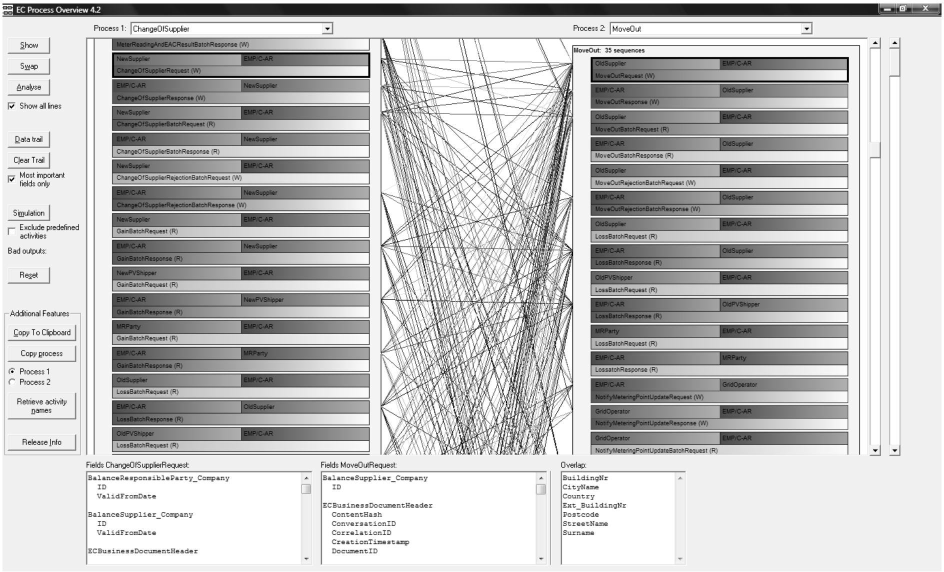
Due to the complexity of the comparisons done in the analysis, a software tool has been developed to automate the simulation of all execution orders. This tool is capable of determining overlap between datafields used by two processes and execute a combinatorial analysis to identify potentialy erroneous outcomes. As a main result, this software tool enables to assess the severity of the interference between two processes.

The analysis tool shows a graphical representation of two selected processes, including individual activities. Potential overlap is indicated by activities marked in red and connection lines between all interfering activities.

For each process an activity can be selected, to show the inputs and outputs of these activities and their specific overlap. In Figure 4.10, an example is provided of the data overlap of the *Change of Supplier* process and the *Move Out* process in the EC case.

After this first analysis, the overlapping datafields are identified for all activities.The severity of the overlap between a supplier change and an address change is immediately suggested by the haywire of lines between the two processes and the large amount of interfering activities.

Prior to starting the simulation, a number of options can be selected to reduce the



Process 1: ChangeOfSupplier

Process 2: MoveOut

☑ Show all lines

☑ Most important fields only

☐ Exclude predefined activities

● Process 1
○ Process 2

Fields ChangeOfSupplierRequest:

```
BalanceResponsibleParty_Company
  ID
  ValidFromDate

BalanceSupplier_Company
  ID
  ValidFromDate

ECBusinessDocumentHeader
```

Fields MoveOutRequest:

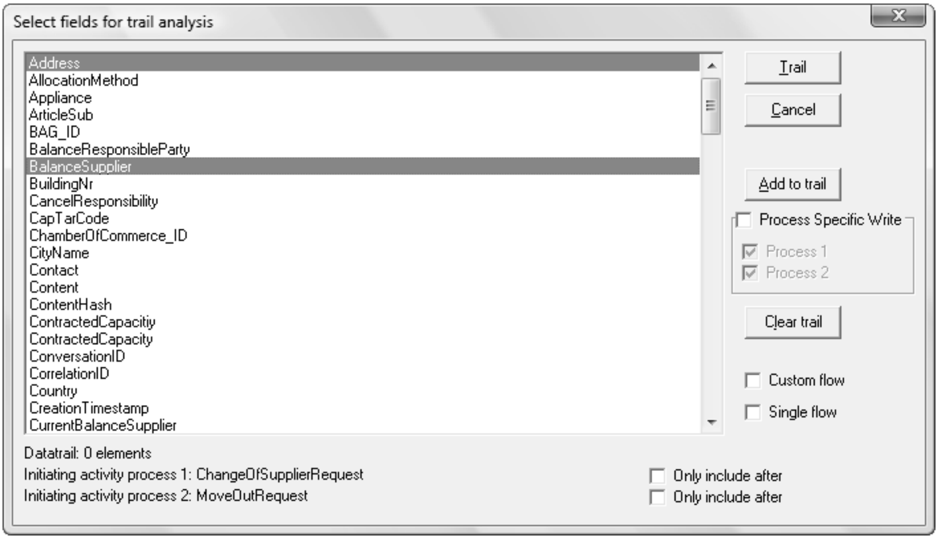
```
BalanceSupplier_Company
  ID

ECBusinessDocumentHeader
  ContentHash
  ConversationID
  CorrelationID
  CreationTimestamp
  DocumentID
```

Overlap:

```
BuildingNr
CityName
Country
Ext_BuildingNr
Postcode
StreetName
Surname
```

Figure 4.10: Screenshot showing overlap in the EC case.



Address
AllocationMethod
Appliance
ArticleSub
BAG_ID
BalanceResponsibleParty
BalanceSupplier
BuildingNr
CancelResponsibility
CapTarCode
ChamberOfCommerce_ID
CityName
Contact
Content
ContentHash
ContractedCapacitiy
ContractedCapacity
ConversationID
CorrelationID
Country
CreationTimestamp
CurrentBalanceSupplier

☐ Process Specific Write
☑ Process 1
☑ Process 2

☐ Custom flow
☐ Single flow

Datatrail: 0 elements
Initiating activity process 1: ChangeOfSupplierRequest
Initiating activity process 2: MoveOutRequest

☐ Only include after
☐ Only include after

Figure 4.11: Screenshot showing selection of fields to incorporate in analysis.

amount of activities according to the criteria outlined above. By default, only those activities are incorporated that use overlapping data. The datafields to be taken into account in the analysis can be specifically selected (Figure 4.11). A datafield can

Function	**Description**
Main form	
Show	Shows selected processes with activities. When selecting activities, the corresponding datafields of that activity are shown.
Analyze	Analyzes the selected processes for data overlap. All activities with data overlap are marked red. Upon selection of a red activity, connecting lines with other activities indicate the overlap. If option *'Show All Lines'* is selected, all overlap between the processes is shown.
Data Trail	For a selected activity datafields can be selected to identify which activities use those datafields as well. The option *Most Important Fields Only* allows to filter the list to represent important fields only.
Simulation	Opens the simulation form (see Figure 4.11 to select the fields to incorporate in the simulation). Option *Exclude Predefined Activities* allows to exclude certain activities from the analysis. These activities can be marked in the analysis database. *Bad Outputs* shows the amount of unique bad outputs identified so far.
Copy to clipboard	Copies graphical image of the processes to the memory.
Copy process	Copies the list of activities for the selected process to the memory.
Retrieve act. names	The activity names are retrieved based on a list of activity numbers.
Field selection form	
Trail	Combinatorial simulation is started.
Add to trail	Adds the selected fields to the analysis.
Process specific write	If a datafield is added to the analysis, it can only be changed by the selected process. That is, WRITE activities from the other process will not affect the value of this datafield.
Custom flow	Opens an inputbox to enter a specific combination. The combinatorial analysis starts the simulation at the predefined combination.
Single flow	Opens an inputbox to enter a specific combination. Executes the predefined combination only.
Only include after	Only includes activities in the analysis that occur after the selected activity in the process (including the selected activity).

Table 4.5: Overview of the important functionality of the tool.

be added to the entire analysis, or be limited to a single process. In the latter case, it can only be changed by the selected process. As a result, WRITE activities from the other process will not affect the value of this datafield.

If necessary, the analysis can be started from the first essential WRITE activity in one or both processes. Using this option, all activities before the selected WRITE activity will not be incorporated in the analysis. Although this is not a mandatory step, it may be required to reduce the amount of execution combinations to achieve a feasible execution time of the simulation. An overview of the functionality of the tool is shown in Table 4.5.

4.7 Analysis

In this section, each pair of processes selected in Section 4.4 will be discussed. For each pair, the initial values and desired result will be provided. The desired result is obtained by executing the pair sequentially. In a well-designed process the outcome of either sequence ($Process_1$ prior to $Process_2$ or $Process_2$ prior to $Process_1$) is the same. Furthermore, the analysis tool provides the same results for both sequences.

In addition, an example of an erroneous result will be provided, along with an informal description of the actual events corresponding with that result. The low-level results of the analysis, including the metadata as used by the analysis tool, can be found in Appendix A.

4.7.1 Energy Company erroneous combinations

Change of Supplier – Move Out

In this comparison, the situation is analyzed where a person changes his energy provider and decides to move to a new address at about the same time. Three datafields are traced: *Supplier*, *New Supplier*, and *Address*. These datafields are common for both processes and hold the current energy supplier of the customer, the new energy supplier of the customer, and the address respectively. These datafields are used by all stakeholders shown in Table 4.6.

The desired outcome is obtained by executing *Change of Supplier* and *Move Out* sequentially. The *Change of Supplier* process contains 16 relevant activities that use one of the datafields under investigation. The *Move Out* process contains 10 relevant activities. All 5 311 734 relevant different combinations in the analysis provide a different output than the desired output. In total, 11 239 distinct outputs were identified, different than the desired output. The different outputs were analyzed, to identify *erroneous* outputs, i.e. outputs that have a different origin (as explained in Section 4.5.3). In Table 4.6, an example is provided of such an erroneous output.

Stakeholder	**Supplier**	**New Supplier**	**Address**
CCP	Correct	Correct	Correct
EMP	Different value	Different value	Different value
GridOperator	Different value	Correct	Different value
MRParty	Different value	Different value	Different value
NewPVShipper	Different value	Different value	Different origin
NewSupplier	Different value	Different value	Different origin
OldPVShipper	Correct	Different value	Correct
OldSupplier	Different origin	Different value	Different value
TM2010	Different origin	Correct	Correct

Table 4.6: Erroneous output 1st comparison of the EC case.

In 7 389 cases, the address known to the stakeholder *New Supplier* has a different origin, which implies that the incorrect address is known to the new supplier (as also shown in Table 4.6). When a switch is proposed, the delivery of energy by the desired new energy provider is linked to a certain address (that is, the address of the customer is retrieved and coupled to the connection data). If the address of the customer changes after this part of the process, the address change is updated to the customer data. Consequently, the new address will be correctly updated, but the change of the energy supplier will not be actualized for this customer. Instead, the desired energy supplier change will apply for the old address. As a result, the new inhabitant of the old house of the customer, will have the new energy supplier as requested by the customer (i.e. the previous inhabitant).

It is possible that the wrong address is attached to a certain energy contract. As a result, the inhabitants of both the old and new address will receive the wrong invoice.

Change of Supplier – Meter Change

In this comparison, the situation is analyzed where a person changes his energy provider and his meter is to be changed at about the same time. Three datafields are traced: *Supplier*, *Meter Reading*, and *Address*. These datafields are common for both processes and hold the current energy supplier of the customer, the meter reading used for the final invoice at the end of the contract, and the address respectively. These datafields are used by all stakeholders shown in Table 4.7.

The desired outcome is obtained by executing *Change of Supplier* and *Meter Change* sequentially. The *Change of Supplier* process contains 19 relevant activities that use one of the datafields under investigation. The *Meter Change* process contains 7 relevant activities. Out of 657 799 relevant different combinations in the analysis, 657 780 relevant different combinations in the analysis, provide a different output than the desired output. In total, 3 522 distinct outputs were identified, different than the desired output. The different outputs were analyzed, to identify *erroneous* outputs, i.e. outputs that have a different origin. In Table 4.7, an example is provided of such an erroneous output.

Stakeholder	Supplier	Meter Reading	Address
EMP	Correct	Different value	Different value
GridOperator	Correct	Correct	Different value
MRParty	Different value	Correct	Different value
NewPVShipper	Different value	Correct	Different value
NewSupplier	Different value	Different origin	Different value
OldPVShipper	Different value	Correct	Correct
OldSupplier	Different value	Correct	Different value
PVShipper	Different value	Correct	Different value
Supplier	Different value	Correct	Different origin
TM2010	Correct	Correct	Correct

Table 4.7: Erroneous output 2nd comparison of the EC case.

In 491 cases, the address known to the old supplier has a different origin, which implies that the incorrect address is assigned to the current supplier (as also shown in Table 4.7). In addition, the meter reading known to the new supplier has a different origin in 1 420 cases, which implies that the new supplier has received the wrong meter reading. As a result, the energy consumption calculated for the final invoice may potentially be incorrect, as it is based on the wrong meter reading. Moreover, the invoice may potentially be sent to the wrong address.

> *The possibility exists that the final invoice sent to the customer constitutes the wrong energy consumption and may be sent to the wrong address*

Change Of Metering Responsible – Move Out

In this comparison, the situation is analyzed where the metering responsible is changed for a certain contract and the owner of that contract decides to move to a new address at about the same time. Three datafields are traced: *Current MR*, *New MR*, and *Address*. These datafields are common for both processes and hold the current metering responsible party, the new metering responsible party, and the address respectively. These datafields are used by all stakeholders shown in Table 4.8.

The desired outcome is obtained by executing *Change Of Metering Responsible* and *Move Out* sequentially. The *Change of Metering Responsible* process contains 9 relevant activities that use one of the datafields under investigation. The *Move Out* process contains 6 relevant activities. All 5 004 relevant different combinations in the analysis, provide a different output than the desired output. In total, 272 distinct outputs were identified, different than the desired output. The different outputs were analyzed, to identify *erroneous* outputs, i.e. outputs that have a different origin. In Table 4.8, an example is provided of such an erroneous output.

In 113 cases, the address known at the EMP (Energy Metering Point), Grid operator and MRParty is different than in the sequential case (as shown in Table 4.8), which is implies that the incorrect address is assigned to these stakeholders. As a result, the customer is charged for the consumption at the wrong address and will, therefore, be charged for the wrong amount.

Stakeholder	Current MR	New MR	Address
EMP	Different value	Different value	Different origin
GridOperator	Correct	Correct	Different origin
MRParty	Correct	Correct	Different origin
NewMRParty	Different value	Different value	Different value
OldMRParty	Different value	Different value	Correct
OldPVShipper	Different value	Different value	Correct
OldSupplier	Different value	Different value	Different value
PVShipper	Correct	Correct	Different value
Supplier	Correct	Correct	Different value

Table 4.8: Erroneous output 3rd comparison of the EC case.

The possibility exists that wrong address is used for measuring the energy consumption of the customer.

4.7.2 Telecom Company erroneous combinations

Buy Packages and Options – Close Customer without Freezing

In this comparison, the situation is analyzed where a customer creates new orders on packages and options while his account is closed at about the same time. Three datafields are traced: *AccessADSL*, *AccessDialUp*, and *AccessWiFi*, which hold the status of the services available to the customer. These datafields are common to both processes and are used by all stakeholders shown in Table 4.9.

The desired outcome is obtained by executing *Buy Packages and Options* and *Close Customer without Freezing* sequentially. That is, first the customer creates a new order on a package, next the account is closed. If one or more of these three datafields did not change after the closing of a customer, potentially these services are still available to the customer after the closing of a customer (and, therefore, after terminating the contract). *The Buy Packages and Options* process contains 3 relevant activities that use one of the datafields under investigation. The *Close Customer* process contains 4 relevant activities. All 34 relevant different combinations in the analysis provide a different output than the desired output. In total, 24 distinct outputs were identified, different than the desired output. The different outputs were analyzed, to identify *erroneous* outputs, i.e. outputs that have a different origin. In Table 4.9, an example is provided of such an erroneous output.

Stakeholder	AccessADSL	AccessDialUp	AccessWiFi
BPM Layer	Different origin	Different origin	Different origin
Prov. Interface	Different origin	Different origin	Different origin
Front End	Correct	Correct	Correct
Infranet	Correct	Correct	Correct

Table 4.9: Erroneous output 1st comparison of the TC case.

In 10 cases, the values of the access to the order packages known to the BPM Layer have a different origin, which implies that the incorrect order status is assigned to the BPM Layer. These results can be observed in the processes as follows. The first process (*Buy Pacakages and Options*) is executed and starts with reading the values from the BPM Layer. As shown in Table 4.9, the erroneous combination starts writing data in the BPM Layer and the Provisional Interface with the values originating from the Provisional Interface. Consequently, the final value of the variables are not in line with the desired values and potentially open orders exist after a customer-account is closed.

> *A customer can be closed and still have open orders on packages and options.*

Customer Move – Close Customer at End of Contract Terms

In this comparison, the situation is analyzed where a customer decides to move to a new address while his account is closed at about the same time. Three datafields are traced: *CustomerBlocking*, *Services* and *Address*. *CustomerBlocking* shows if a customer is to be closed and is composed of a group of attributes including blocking flag, blocking remark and blocking date. *Services* holds the products delivered to the customer. These datafields are common to both processes and used by all stakeholders shown in Table 4.10.

The desired outcome is obtained by executing *Customer Move* and *Close Customer* sequentially. That is, first the customer moves, next the contract is ended. The *Customer Move* process contains 3 relevant activities that use one of the datafields under investigation. The *Close Customer* process contains 14 relevant activities. Out of 680 relevant different combinations in the analysis, 677 combinations provide a different output than the desired output. In total, 181 distinct outputs were identified, different than the desired output. The different outputs were analyzed, to identify *erroneous* outputs, i.e. outputs that have a different origin. In Table 4.10, an example is provided of such an erroneous output.

Stakeholder	**CustomerBlocking**	**Servicess**	**Address**
BPM Layer	Different origin	Different origin	Different value
Prov. Interface	Different value	Correct	Correct
Front End	Correct	Correct	Correct
Infranet	Different origin	Different origin	Different value

Table 4.10: Erroneous output 1st comparison of the TC case.

In 175 cases, the value of *Services* known to the BPM Layer has a different origin, which is implies that the incorrect status of the provided services is assigned to the BPM Layer. If a customer is closed, the products of the customer should not be available anymore to the customer. As a consequence, they cannot be moved to a different address. However, if the products are moved to the new address after closing, the products will still exist, while the customer already has closed his account. This is clearly shown by the analysis results, where the value of *Services* differs from the value in the erroneous case (Table 4.10). That is, the services provided to the customer can be changed while the request for closing the account is already initiated.

The possibility exists that a customer account is closed, but also a new order is created to move a Broadband package.

Upgrade/Downgrade/Switch – Upgrade from ADSL to VoIP/IPTV/Broadband

In this comparison, the situation is analyzed where two processes are executed simultaneously to update a package of products. The first process is using the gathered data to downgrade a package, whereas the second is using the same data for upgrading the package to a Broadband package. The datafield *AccountDetails* is traced, which is used for the change in packages and can have different values depending on the package currently used by the customer. This datafield is common to both processes and used by all stakeholders shown in Table 4.11.

The desired outcome is obtained by executing *Upgrade/Downgrade/Switch* and *Upgrade to Broadband* sequentially. That is, first the package is downgraded, next the package is upgraded to broadband. The *Upgrade/Downgrade/Switch* process contains 6 relevant activities that use one of the datafields under investigation. The *Upgrade to Broadband* process contains 5 relevant activities. Out of 462 relevant different combinations in the analysis, 456 combinations provide a different output than the desired output. In total, only 3 distinct outputs were identified, different than the desired output. The different outputs were analyzed, to identify *erroneous* outputs, i.e. outputs that have a different origin. In Table 4.11, an example is provided of such an erroneous output.

Stakeholder	**AccountDetails**
BPM Layer	Correct
Prov. Interface	Correct
Front End	Different origin
Infranet	Different value

Table 4.11: Erroneous output 1st comparison of the TC case.

In 2 cases, the account details known from the Front End have a different origin, which is implies that the incorrect account details are used throughout the remainder of the process.

If the order to downgrade a package could coexist along with an order for the upgrade to another package, uncertain outcomes could occur as it is not clear which of both orders is valid. As shown in Table 4.11, the BPM Layer uses different account details than inserted in the Front End. Consequently, different orders may interfere and result in the different order provided than requested.

It is possible to use the same account data for creating different orders, or even use different account data for different orders of the same customer.

4.8 Validation of results with process experts

The results of the analysis were validated by means of interviews with the process experts at the EC and the TC. The validation consisted of informal interviews with

4 different process experts. First, the individual processes obtained by the analysis were shown to process experts of the EC, to verify the process representation used by the analysis with the execution of these processes in reality. It showed that all processes as represented were reflecting the execution of business processes reality.

Subsequently, the analyzed process pairs were assessed with the execution of business processes in reality. That is, the concurrent execution as represented by the analysis tool was validated with the potentiality of such a co-occurrence in reality.

Finally, the organizations awareness of each troublesome case was evaluated. The interviews with the process experts clearly revealed the business nature of the problem, as the majority of the results was unknown. The most characteristic example of such an unknown case is the parallel execution of a Supplier Change and a Move Out.

Most of the problems emerging from the overlapping scenarios concerned customer data or connection data without resulting in failing processes. Consequently, the problems primarily affected the external stakeholders (customers), whereas they did not directly affect the internal resources. As a result, most of these scenarios were past the awareness of the organization and no mechanisms or procedures were in place to prevent, correct or identify these errors. However, the process experts were aware of a large number of customer complaints, which could now be explained through the scenario's obtained from the analysis.

Two of the severe cases were within the awareness of the organizations. These processes were equipped in an ad-hoc manner with various mechanisms designed to minimize the risk for these errors. The most typical cases of interference were intercepted by custom-built triggers to either enforce alignment between the processes or provide a process lock. That is, one of the processes is not allowed to proceed until completion of the other process or not allowed to start at all.

4.9 Conclusion

The analysis showed that concurrently executed processes indeed may interfere in practice. More specifically, almost *all* different activity orders involving important data are causing erroneous outcomes. Although the total amount of activities (and, therefore, different combinations if all activities are taken into account) is much higher, the combinations used in the analysis only comprise relevant activities with respect to the data used. As a result, the analysis clearly shows the effect of different execution orders of relevant activities using essential data.

The validation with process experts revealed that the unknown problems as indicated by the analysis tool are common practice in reality as well. The amount and severity of the identified interference confirms the frequency of occurrence of the problems as well as the according relevance for organizations.

Compared to other methods, this is a rather lightweight method. That is, it does not require the availability of a formal representation of the business process. Instead, it is applicable using semi-structured process documentation, providing results that are legible by users without in-depth knowledge of implementation specifics.

However, the application of the methodology does not reveal whether the problems can be prevented by means of, for example, a better software implementation. Rather, application of the methodology identifies the potentially interfering processes. Moreover, it provides insight in the severity of potential interference between concurrently executed processes, which provides the opportunity to resolve or prevent these situations in the Enterprise Information System.

Correspondingly, the interference found in the analysis of EC and TC is not a result of a poor software implementation. The analysis has been performed independently from any implementation. In this respect, this analysis has gone beyond past research, by analyzing the process flow along with the information required in each of the distinct activities. The results of the application of the methodology to the cases clearly show the importance and relevance of these business problems.

The methodology showed its ability to efficiently provide a representative and valuable insight in the interference between concurrent processes and the potential disruptions emerging. Using this insight, additional measures can be taken in order to identify and resolve potentially erroneous situations. In Chapter 5 and Chapter 6, a framework is designed to prevent process interference by awareness of process dependencies and automatic execution of compensation activities.

CHAPTER 5

Dependency scopes and intervention processes

5.1 Introduction

Runtime handling of interference is required, in order to identify and resolve potentially erroneous situations. In this chapter, process interference is prevented by awareness of process dependencies and automatic execution of compensation activities. Dependency scopes are introduced to represent the dependencies between processes and data sources. In addition, intervention processes are developed to repair inconsistencies using dynamic reconfiguration during execution of the process. A business process supporting the WMO law is examined, to demonstrate the proposed solution and to show feasibility of the approach.

First, the definition of the basic concepts is provided, where the approach for BP repair is built upon. In this chapter, it is not necessary to provide the full semantics of a business process to describe the basic concepts of dependency scopes and intervention processes. Although a more elaborate, formal BP definition will be provided in Chapter 6, for readability a more informal working definition of a business process (BP) is provided in this chapter. The concept of a business process is defined here as follows:

Definition 5.1.1 (Business Process)**.** *A business process is a set of linked activities, constructors and process variables that collectively realize a business objective or a policy goal, where:*

- *Each activity is an atomic piece of work representing an interaction with some service.*
- *Constructors represent the flow of execution, e.g. sequence, choice, parallelism, join synchronization. These constructors have well-defined semantics, e.g. defined in (Van Der Aalst et al., 2003b).*
- *A process variable is a variable over an arbitrary domain, which is typically mapped into input/output parameters of activities (services).*

Definition 5.1.2 (Sub-process)**.** *A sub-process is a business process that is enacted or called from another (initiating) business process (or sub-process), and which forms part of the overall (initiating) business process (WfMC, 1999).*

The process definitions presented above are not new. They have been implemented in different workflow and business process management systems, e.g. using BPMN, or a BPEL notation.

Definition 5.1.3 (Volatile process variable)**.** *A volatile process variable is a process variable that can be changed externally during execution of the process.*

In Figure 5.1, two processes are presented. The decision made in Process 1 is based on the value of process variable d. That specific decision determines whether activities A1 and A2 are executed or rather A3 and A4. If d is changed by another process (e.g. Process 2) during execution of A2, this may have consequences for the decision made. That is, as a result of the data change, currently the wrong branch of activities is being executed. In such a situation, the execution of A2 needs to be cancelled and followed by compensating activities to compensate A2 and

A1. Subsequently, the process should continue at A3. Therefore, it is desirable to know what activities are implicitly relying on that process variable (d). Furthermore, these activities should be notified if that data has changed, even if those changes happened externally to the process being currently executed.

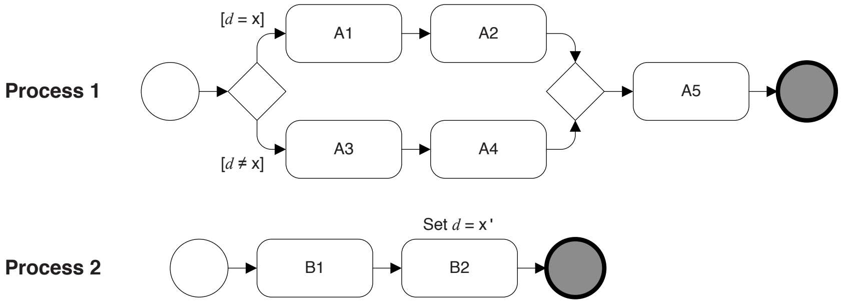


Figure 5.1: Two business processes with concurrent data modification.

5.2 Dependency scopes

To identify the specific part of the process that depends on certain process variable, we introduce a notion of dependency scope. Although a more elaborate, formal definition will be provided in Chapter 6, for readability a more informal working definition of a dependency scope is defined here as follows:

Definition 5.2.1 (Dependency Scope)**.** *A dependency scope (DS) is a structurally correct subset of the business process, in which the activities are implicitly or explicitly relying on the accuracy of a volatile process variable accessed in the first activity of that set.*

During the execution of the entire DS, the process variable is assumed to remain unchanged (or within a certain range of values) by an external process. Note that this definition implies that an update of this process variable by the process will end the DS of that particular process variable, whereas it may start a new DS. In Figure 5.2, Process 1 is represented with a corresponding dependency scope. At an instance level, a DS can be active or inactive. It is activated when the first activity is started, which is part of the set that defines the DS. It is active as long as an activity is executed that belongs to the DS. If the last activity of the set of consecutive activities is finished, the DS switches to be inactive.

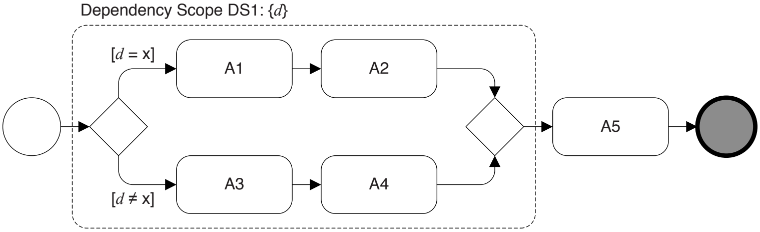


Figure 5.2: Business process with a dependency scope definition.

5.3 Intervention processes

If the variable d is changed by a concurrent process (e.g. Process 2 in Figure 5.1) while DS1 is active, DS1 is notified of that change. If the change of d occurs during the execution of A2, A1 has already executed. Consequently, some action is required to resolve the conflict caused by the change of d: the process should restart again from the decision point prior to activity A1, which requires both A1 and A2 to be rolled back to the initial state. This rollback may in some cases be provided by a number of alternative activities to be executed before starting A3. For example, the state of Process 1 is undesirable from a business perspective, due to the incorrect decisions made as a result of the change of d. One or more repairing activities should be interposed between A2 and A3, to recover Process 1 to a consistent state that corresponds to reality again.

Definition 5.3.1 (Intervention Process)**.** *An intervention process is a sub-process that is linked to a DS, comprising a set of compensation activities, which together restore the consistent state of a business process. An intervention process has the following properties:*

- *A condition over the set of data elements D of the DS determines when the set of compensation activities needs to be executed.*
- *If the condition is true then the currently executed activity in the DS is stopped and the compensation process is executed.*
- *The last activity provides a re-entry point in the business process.*

Figure 5.3 shows a sequence of compensating activities, which is defined as an intervention process.

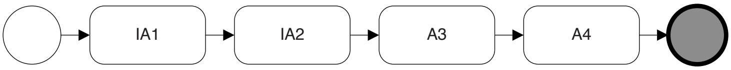


Figure 5.3: Specification of intervention activities.

The activities required to restore consistency may vary, even concerning the same volatile process variable. However, if more intervention processes are connected to one DS, then the conditions should be mutually exclusive. In addition, the activities required to restore consistency may vary between processes. In some cases, it may be sufficient to update the process variables in the currently executed activity and proceed, whereas in a more severe case the activity needs to be cancelled and to process should be resumed with another activity. An example of the process including both a DS and an inserted intervention process is shown in Figure 5.4. This solution allows for execution without manual process reconfiguration. As a result of the activation of DS1, the activity currently being executed (A2) is halted. Next, the process is continued at the Continue mark in Figure 5.4. This will start the execution of the intervention activities. Note that IA1 and IA2 are not necessarily cancellation or compensation activities, but may also be additional activities that are required to finish the process regularly. After the intervention activities have been finished, the process proceeds after DS1 in the regular process flow (A5). As a result, A3 and A4 are part of the intervention process.

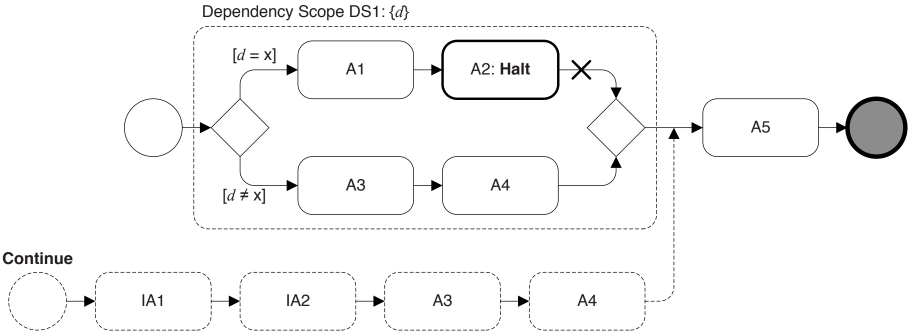


Figure 5.4: Business process with dependency scope and connected intervention activities.

The concepts described above prevent the process designer from being forced to check the value of the condition after every activity within the DS. That is, in order to predefine the error-handling in case of process interference without the presented concepts, for every activity the values of volatile process variables have to be tested. A (simplified) example of such an undesirable situation is represented in Figure 5.5. In more complex business processes, this would require a high amount of checks predefined in the business process. It is to be expected that this way of overcoming interdependency issues will strongly increase the complexity of each process model and, accordingly, result in a cascading change after the model is to be updated.

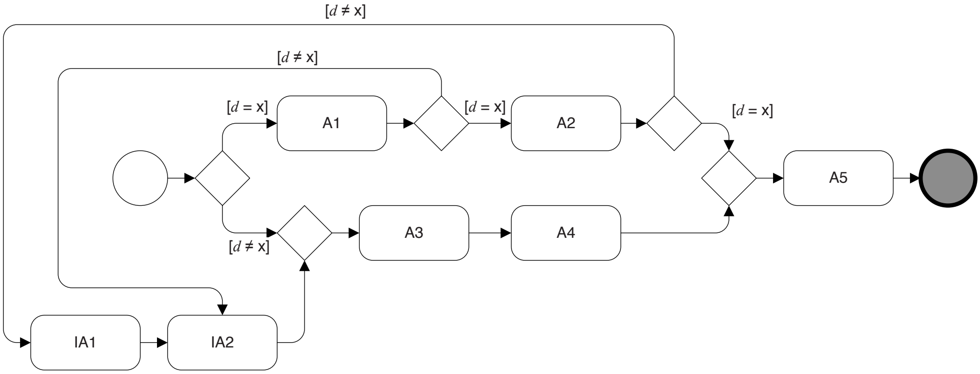


Figure 5.5: Alternate solution to resolve dependencies.

5.4 WMO dependency scope example

The request for a wheelchair or a home modification may take up to 6 weeks until the delivery of the provision. These processes depend on the correctness of some process variables. However, these process variables may be changed by another process running in parallel, independent of the WMO process, and are, therefore, volatile. A change in either of these volatile process variables may potentially have negative consequences for the WMO process, due to its dependencies on those variables, and result in undesirable business outcomes. Consequently, changes in these variables pose a potential risk of interference.

For instance, the activities after the decision until delivery are strongly depending on the accuracy of the citizen's address. That is, the requirements of the wheelchair not only depend on the citizen, but also on the residence as this may pose some constraints to e.g. the width of the wheelchair. Consequently, an address change after “Acquire requirements” might result in a wheelchair that does not fit the actual requirements. Similarly, if the citizen moves to a nursing home after “Check tender with decision”, the home modification is not necessary anymore. However, the supplier is not notified of this address change and the municipality is notified through a different process, which is external to the WMO process. As a result, unless some action is taken to cancel or update the order, the WMO process will proceed with the home modification. In addition to “address”, the process depends on the medical condition of the citizen, after executing the home visit and obtaining the medical advice. If the condition of citizen deteriorates, potentially the provision needs to be adjusted. If, on the other hand, the condition improves, the provision may be no longer necessary.

In order to guard for changes to the volatile process variables, DSs can be defined covering a section of the process for which such a change poses a potential risk of interference. In Figure 5.6, a part of the process is annotated with the appropriate DSs. The section covered by DS1 relies on the accuracy of the address as well as the medical condition of the citizen, while the section covered by DS2 relies on the accuracy of the WMO eligibility criteria. That is, if the legal criteria that are relevant for the used contract have changed, this might affect the order itself, or the potential suppliers that are participating in the tender procedure. Finally, the section within DS3 depends on the address and the medical condition of the citizen as well, however is separate from from DS1 because of the syntax of the BP. If a DS is triggered by an external change on its process variable, potentially some recovery activities need to be executed to restore consistency.

5.5 Required intervention processes

The required IPs may differ for each situation. For example, if the address change is detected before the order for a wheelchair is sent to the supplier, it is sufficient to execute the IP as shown in Figure 5.7a. However, if the order is already sent to

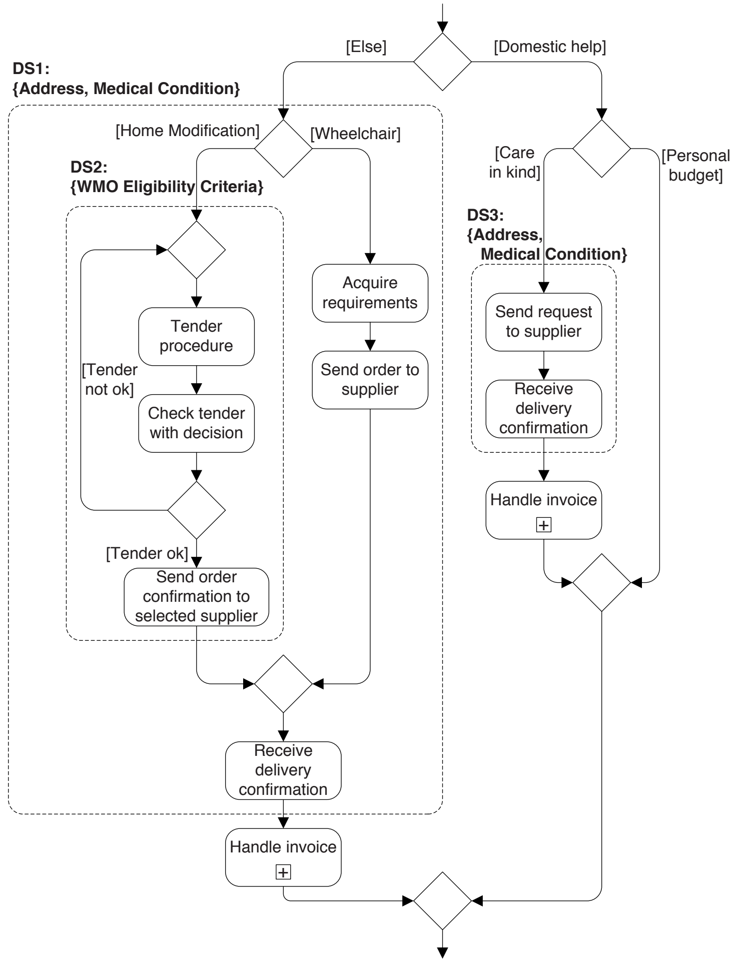


Figure 5.6: Dependency scopes in the WMO process.

the supplier, some additional activities are required (Figure 5.7b). First of all, the current order should be put on hold. After acquiring the requirements again, it is evaluated whether there is a change. If not, the order can be resumed, otherwise the old order should be cancelled and a new order should be sent.

Similarly, in case of home modification the IP also depends on the state at which the address change occurs. If the address changes before the order is sent, it is sufficient to execute the IP as represented in Figure 5.7c. Since the specifications on the order directly rely on the address, a change of address implies a cancellation of

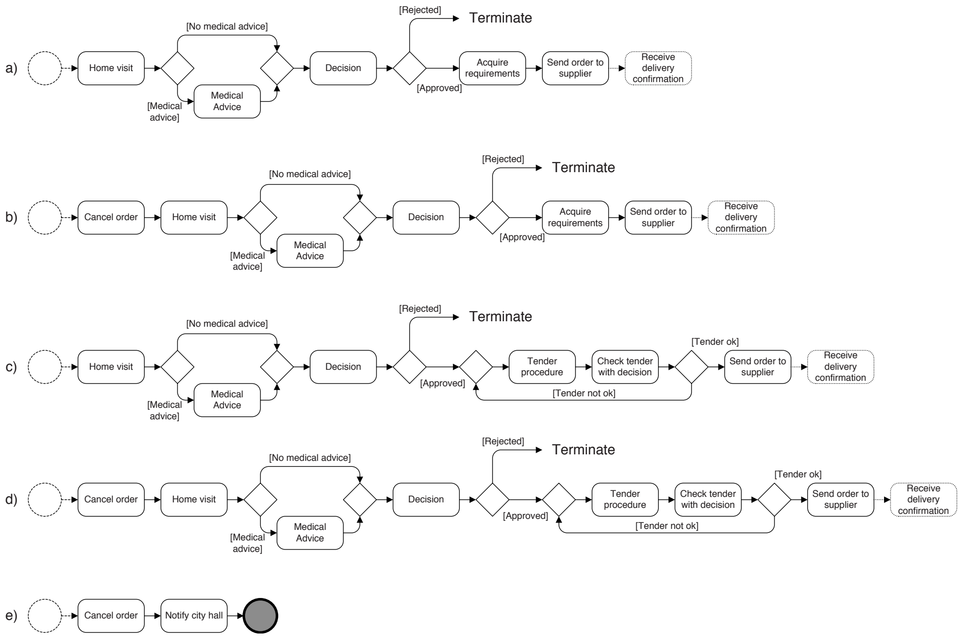


Figure 5.7: Required intervention processes corresponding to DS1, in case of an address change

the order in all cases, if an order has already been sent. The remainder of the IP is identical, as shown in Figure 5.7d. As opposed to the case of a wheelchair request, the decision for the home modification depends explicitly on the physical properties of the house itself. As a result, an address change may have its effect on the decision, as the home modification may no longer be necessary in the new situation (e.g. a request for an elevator will not apply if the citizen moves to a single-floor residence). Therefore, the decision should be revised if the new situation differs from the old situation, upon which the initial request was based. If the decision is again positive, the IP proceeds similarly to the original BP. However, these examples assume that the citizen moves *within* the municipality (in our example this is 'Groningen'). If the citizen has moved to another municipality, the entire process should be cancelled, regardless of the requested provision, as each municipality has its own policies and procedures (Figure 5.7e).

5.6 Implementation

To show the feasibility of the approach, a prototype has been implemented on top of a business process management platform (BPMP), which is a result of a joint work with Pavel Bulanov and the implementation is mainly performed by him. This BPMP adheres to modern change management techniques, such as case handling and process inheritance, thus providing advanced runtime reconfiguration abilities. A detailed discription of the BPMP is provided in Chapter 7. The prototype adds dependency scopes over existing business process models, and maps each of the defined dependency scopes to an appropriate trigger.

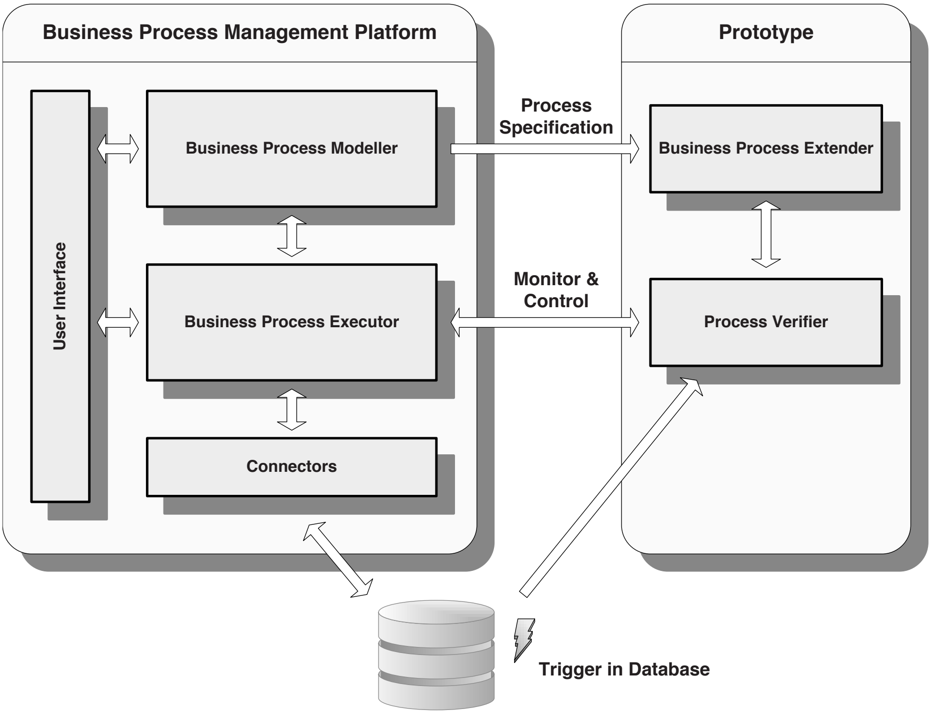

Business Process Management Platform
Prototype
User Interface
Business Process Modeller
Business Process Executor
Connectors
Process Specification
Monitor & Control
Business Process Extender
Process Verifier
Trigger in Database

Figure 5.8: Architectural overview of the prototype.

Figure 5.8 depicts the architectural overview, where the left box represents the simplified BPMP architecture, while the right box represents the structure of the prototype itself. In the BPMP box, the major parts are the *Business Process Modeller* and *Business Process Executor*. The former provides visual process design facilities, whereas the latter is responsible for process monitoring and execution. Connectors at the bottom of the BPMP box provide communication with external data, e.g. databases, or data provided by external services. Finally, the user interface provides means for interaction with users, and is usually represented by a web-based application.

In the prototype box, the *Business Process Extender* parses the process specification in order to extract dependency scopes. Such dependency scopes can be designed using standard BPMP process design facilities, and the information about dependency scopes is saved along with the specification of the process in the internal process repository. Subsequently, the process specification can be retrieved via the BPMP public API.

When the data modification occurs, a trigger is fired, which passes the corresponding information to the *Process Verifier*. This verifier has access to the information about existing business processes and their dependency scopes, which is provided by business process extender. Based on the information about the processes being currently executed by the BPMP, the process verifier makes the decision whether or not to stop process(es) and fire the appropriate intervention process(es). In order to support decision making, additional information must be associated with every dependency scope, such as a table in the database, the criteria to find a row in the table, and the criteria to identify which changes in the data are significant.

In Figure 5.9, the WMO process is modeled using the process designer with the business process extender on top of it. Two nested dependency scopes (DS1 and DS2) are specified. DS1 is assigned to Address, whereas DS2 is assigned to WMO eligibility criteria. DS1 is associated with the table Citizens in the underlying database and, whenever the Address is changed, the corresponding intervention process is executed.

For example, consider the situation where a wheelchair is requested. Both Acquire detailed requirements and Send order to supplier were executed, and Delivery is about to be executed. If the address of the citizen is changed, potentially a erroneous situation may occur. Since the process is now within DS1, some intervention

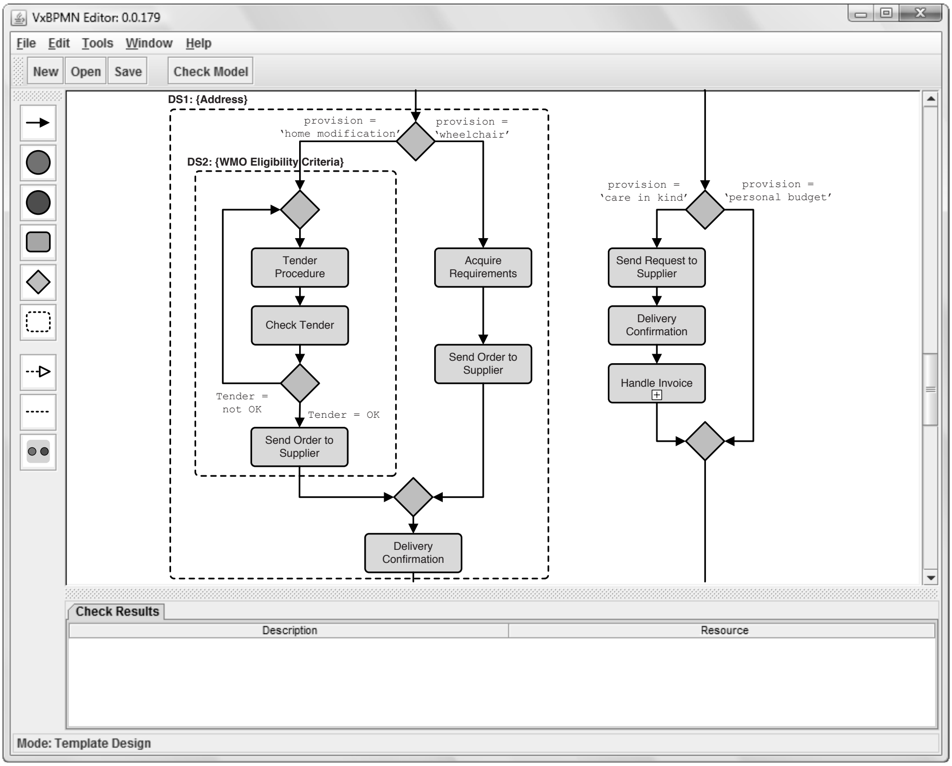


Figure 5.9: Screenshot of dependency scope implementation within the BPMP.

actions must be undertaken.

The sequence of actions in this case is the following:

- The process verifier is called with the information that a row is modified in the Citizens table.
- The dependency scopes associated with the Citizens table (which is DS1) are identified.
- The currently running process instances that are now inside that scope are fetched. There is one running process instance, and the activated dependency scope is DS1.
- A check is performed to verify if the data modification is significant for the

process instance under investigation (since the address has been changed, the modification is significant).

When all conditions are met, the intervention process is automatically executed as follows:

- The original process is stopped.
- The compensation process assigned to DS1 is executed, as shown in Figure 5.7b.

A situation might occur that does require intervention, but the predefined intervention processes attached to the dependency scope do not apply for this particular situation. In these cases, two possible solutions can be suggested. First, the process can be paused and require a human decision on how to proceed. Second, a rollback can be executed. This is, however, the least desirable solution, especially in processes with a long lead time.

5.7 Developed concepts and required patterns

The existing patterns concerning representation and utilization of data within processes (Russell et al., 2004) offer a wide range of data constructs and informational concepts that a business process engine is able to capture. Similarly, dependency scopes and intervention processes require specific ways data is represented and utilized in processes.

As such, the required constructs for data and data passing can be defined as data patterns for both dependency scopes and intervention processes, which is very similar to the data patterns defined by Russell et al. (2004). Following the same notation, we extend the available data patterns with *dependency scope data* and an *incoming change event* pattern.

Pattern (Dependency scope data)
Description: Data elements can be defined, whose new values are notified to a

dependency scope upon a change of that data element.

Example: The address variable used in DS1 of Figure 5.6.

Motivation: To provide support for awareness of external data changes in dependency scopes. Typically these data elements will be used for monitoring the changes of that variable by other processes.

Implementation: The definition of dependency scope data elements requires the ability to define the portion of the process (i.e. the dependency scope) to which the data elements are bound. The data elements are accessible by the dependency scope during the execution of the activities in that dependency scope. The activities of an intervention process also belong to the dependency scope, which implies that external data changes are taken into account during execution of an intervention process as well.

Pattern (Data Transfer by Value - Incoming change notification)
Description: The ability of a dependency scope to receive incoming change notifications of a data element with an update of its value accordingly.

Example: If the address changes during any of the activities in DS1 (Figure 5.6), DS1 will be notified of that change.

Motivation: Under this scenario, the values of data elements are passed from the process executor to the dependency scope.

Implementation: This approach to data passing is commonly used for communicating data elements upon their change to the respective subset of the process (i.e. the dependency scope) that is relying on the value of that data element. Based on this change, the process verifier can decide whether or not to execute an intervention process.

Although we do not use the pattern definitions provided above, it is useful to present our work in terms of data patterns defined by Russell et al. (2004), as it will help independent researchers to have a unified view over different approaches to business process modelling and design.

CHAPTER 6

Automated intervention process generation

It becomes evident from the example that even for a small DS, the complexity and workload required for specifying the IPs cannot be underestimated. Manual IP design is prone to oversights of possible situations that may arise: different IPs are required not only depending on the current state, but also on the actual value of the modified variable. As a result, for each possible state in a DS and type of change to the modified variable, a different IP may be required. Moreover, since the same BP may be used by more than one municipality, different IPs have to be specified for each of the different cases, as they may have access to different compensation services or comply with different rules.

Consequently, a mechanism is developed to automatically generate the IP based on the DS, the current state, and the value of the volatile process variable. In this section, the architecture of the framework supporting IP generation is presented. Subsequently, each component of the architecture will be discussed in detail.

6.1 Architectural overview

Figure 6.1 provides an overview of the main components of our framework, along with their basic interactions. A *Process Modeller* (PM) is used to assist with the task of the graphical modeling of the BP, providing a selection of standard control blocks like sequence, flow, switch etc., and design tools for modeling DSs, in accordance with their definition to be provided in Section 6.2. DSs include the specification of some high-level *goals* of declarative nature, which have to be fulfilled by the respective intervention process in case the conditions indicating an inconsistency are fired.

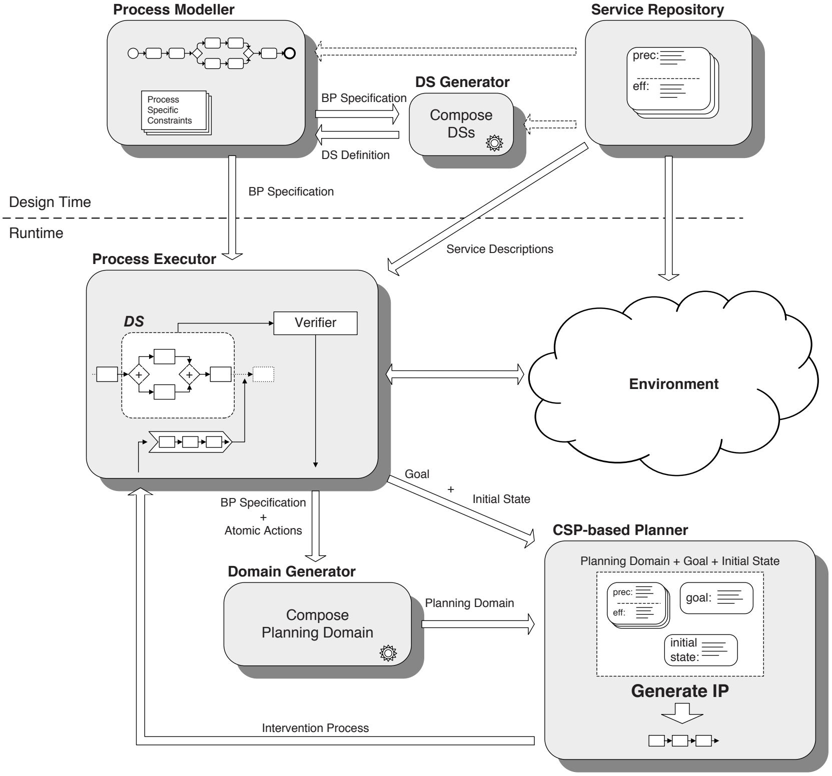


Figure 6.1: Main components of the framework and their basic interactions

The BP modelled by the PM uses activities that are available in the *Service Repository* (SR) by means of service operations. The SR keeps a list of service instances (providers) that offer a set of service operations. Each service instance implements a service type, which specifies the interface of the service captured by some extra semantics. These semantics allow each service operation to be represented as a planning $action$, reflecting its functional behaviour in terms of preconditions and effects, which are necessary for enabling the automatic generation of intervention processes. A subset of the service operations are referenced by the BP specification, whereas operations offered by other service instances can be marked as pertinent compensation actions, and can become part of an IP if necessary.

The required DSs are discovered automatically in the *DS Generator*. The DS Generator automatically computes the appropriate sections in a BP that should be covered by a DS, based on the BP specification (obtained from the PM) and some semantics regarding the input-output and the internal state variables of the service operations (as defined in the SR). The resulting DSs are represented in the PM.

The *Process Executor* (PE) is responsible for executing the BP step by step (i.e. the normal course of events as specified during design-time), and takes care of discovering, binding and invoking the respective service operations residing in the *Environment*, according to their specification as included in the SR. Some of the variables describing the state of the environment can be directly changed by the process being executed by the PE, through the invocation of services it has access to, or can be modified by some external process. In the latter case, the PE receives a modification event, and updates its current internal state accordingly. In addition to process execution, the PE supports the use of DSs. Before execution of each activity, the PE checks whether the current state indicates a modification of the volatile variables that are guarded by a DS that covers this activity. If so, it verifies whether any of the conditions specified in the DS hold. If a condition holds (e.g. the new address is outside the current municipality), then the PE interrupts the execution and invokes the *AI Planner*. The AI Planner requires as input (i) the Planning Domain (ii) the initial planning state (i.e. the values of all process variables at the current execution step and a set of variable interdependencies), and (iii) the goal describing the desired properties to it be achieved (e.g. a notification should be sent to the city hall). Before explaining the AI planner in more detail, we discuss the notion of a Planning Domain.

The Planning Domain is computed by the *Domain Generator* (DG) only once for a certain process instance, the first time that the PE identifies the need for automatic IP generation. In order to form the Planning Domain, the PE passes the *Atomic Actions (AA)* and the BP specification (provided as output by the PM) to the DG. The AA represent the BP-pertinent action descriptions as kept in the SR (i.e. the ones referenced by the BP along with the compensation operations). Given these two inputs, the DG can generate the encoding of the Planning Domain, by enriching the generic action descriptions of the AA with extra preconditions and effects that reflect the BP-specific interdependencies between the actions (e.g. sequence, flow and switch).

Given the Planning Domain, the initial state and the goal, the AI planner generates the appropriate IP that achieves the associated goal. The generated IP is then returned to the PE. After the execution of the IP, the PE either proceeds with the execution of the original BP, starting from the state right after the triggered DS (as in Figures 5.7a-d, where the original BP execution resumes after ”Delivery”), or aborts if the IP leads to a state that indicates the termination of the BP (as in Figure 5.7e). If the former is the case, potential branches that were running in parallel are also resumed from the point they were interrupted, otherwise the entire process is interrupted. In the case of nested DSs, as for example DS1 and DS2 of Figure 5.6, the PE checks first whether the conditions specified by the outermost DS are true, and if not, it proceeds by checking the inner DS. The generated IP is executed within the scope of the DS it was triggered from and the parent DSs. Consequently, variable modifications that are received *during* the execution of an IP are covered by the same DSs that covered the activity in the original BP, before the planner was invoked. If no plan can be found, i.e. there is no way to overcome the inconsistencies caused by the volatile variable modification using the activities it has access to, then the BP is canceled, and a request for manual inspection is issued.

The AI planner will be discussed in detail in Section 6.2.3 and Section 6.3, while the algorithm of the DS generator is discussed in Section 6.4. The implementation of the PE and the PM is presented in Section 7.1.

6.2 Basic concepts

In order to automate the task of intervention process specification, the original BP should be represented in a format which constitutes the appropriate semantic annotations. These annotations comprise the demarcation of the dependendency scopes along with their accompanying goals and the formalization of the participating activities in terms of preconditions and effects. The BP-specific information (concerning its structural constituent elements) is kept separate from the generic, BP-independent service descriptions. The generic service descriptions are maintained in a separate repository and can be referenced by different BPs. The basic syntactic structure of the BP builds upon the standarized executable language for describing BPs with web services, WS-BPEL. In this section, the definition of the basic concepts is provided, where the approach for BP repair is built upon.

6.2.1 Business process

First, we define the Service Repository consisting of a set of service type descriptions and a set of service instances, which “implement” some service type. A service type comprises the semantics which represent the logic of the provided functionalities. Service instances refer to some concrete service offered by by a specific provider, which comforms to a service type (since usually there are many functionally equivalent providers). The semantic markups defined in the service types are necessary in order to automate the task of IP generation. They are expressed in terms of *preconditions*, which model the propositions that have to hold in the current state for an activity to be executed, and *effects*, which formulate how variables are changed by the execution of the activity. The service type descriptions are based on an IOPE (Input Output Preconditions Effects) model, which is followed by established Web Service semantic languages like WSDL-S[1] and OWL-S.

Definition 6.2.1 (Service Type (st))**.** *A service type is a tuple $st = (stid, O, SV)$, where $stid$ is a unique identifier, O is a set of service operations, and SV is a list of variables, each ranging over a finite domain. These variables correspond to state*

[1] www.w3.org/Submission/WSDL-S

variables internal to the service, whose value can be changed by the operations of the service.
Each service operation $o \in O$ is a tuple $o = (id(o), in(o), out(o), prec(o), \mathit{eff}(o))$ where:

- $id(o)$ *is the identifier of the operation.*
- $in(o)$ *is a list of variables that play the role of input parameters to o, ranging over finite domains.*
- $out(o)$ *is a list of variables that play the role of output parameters to o, ranging over finite domains.*
- $prec(o)$ *is a set of preconditions and $\mathit{eff}(o)$ a set of effects (as defined in Definition 6.2.8 with $Var = in(o) \cup$).*

Definition 6.2.2 (Service Instance (si))**.** *A service instance is a tuple $si = (iid(si), st(si))$, where:*

- $st(si)$ *refers to the identifier of the service type $st \in ST$ this instance is compliant with.*
- $iid(si)$ *is the instance's unique identifier. For each pair of service instances si_1, si_2 that have the same service type identifier $st(si_1) = st(si_2)$, $iid(si_1) \neq iid(si_2)$.*

Definition 6.2.3 (Service Repository (SR))**.** *A Service Repository $SR = (ST, SI)$ is a storage, which keeps a set of Service Types ST and a set of Service Instances SI.*

The SR plays the role of a pool of service types and instances, which are used as the building elements of different process specifications. In the following, the definition of a Business Process (BP) is provided, which includes the basic activities and control structures such as sequence, flow and switch. This is an elaborate and formal definition of the BP, as initially defined Chapter 5. The BP is enriched with DSs, which also constitute parts of the process. Although the WMO process (Figure 3.6) is represented in BPMN-notation for readability reasons, the BP specification used in this paper is block-structured (Ouyang et al., 2006; Kopp et al., 2008), and is based on the basic constructs of BPEL. The syntax of the BP is well-defined and unambiguous, so that they can be directly executed by the Process Executor (see Section 7.1.2) and automatically transformed to a representation usable by the planner. The representation is ultimately a tree structure where a block can have other blocks as children, and for each block its parent can be obtained. The definition is recursive, so that control structures and DSs can be nested within each other.

Definition 6.2.4 (Logical Condition (C))**.** *A logical condition C conforms to the following syntax:*

$C \quad ::= prop| \wedge_j C_j| \vee_j C_j)|\neg C_j$

$prop \quad ::= var \circ value \;\mid\; var1 \circ var2 \;|(var1 \diamond var2) \circ value$

where:

- $var, var1, var2$ *are variables ranging over finite domains.*
- val *is some constant belonging to* var*'s domain.*
- $\circ$ *is a relational operator (*$\circ \in \{=, <, >, \neq, \leqslant, \geqslant\}$*).*
- $\diamond$ *is a binary operator (*$\diamond \in \{+, -\}$*).*

Definition 6.2.5 (Business Process (BP))**.** *Given a service repository* SR*, a business process is a tuple* $BP = (PV,\ E)$*, with* E *being a process element* $E = ACT \mid SEQUENCE \mid FLOW \mid SWITCH \mid REPEAT \mid WHILE \mid DS$*, where:*

- $PV = PV_i \cup PV_e$ *is a set of variables ranging over finite domains.*
  - PV_i *is a set of internal variables, which are BP-specific. A subset of these variables are passed as input parameters to the entire BP, and can be initialized with specific values at execution time.*
  - PV_e *is a set of external variables, which refer to state variables declared in the* SR*. An external variable* $v \in PV_e$ *is a reference* $stid.iid.vid$*, where* $stid$

is the identifier of a service type $st = (stid, O, SV) \in ST$, iid is the identifier of a service instance $si = (iid, stid) \in SI$, and vid is the identifier of some state variable $v \in SV$.

- *ACT is a process activity, as defined in Definition 6.2.6.*
- *$SEQUENCE$ represents a totally ordered set of process elements, which are executed in sequence: $SEQUENCE\{e_1 \ldots e_n\}$, where $e_i \in E$.*
- *$FLOW$ represents a set of process elements, which are executed in parallel: $FLOW\{e_1 \ldots e_n\}$, where $e_i \in E$.*
- *$SWITCH$ is a set of tuples $\{(c_1, e_1), \ldots, (c_n, e_n)\}$, where $e_i \in E$ and c_i is a logical condition C, with all variables $\in PV$. All c_i participating in a $SWITCH$ are mutually exclusive, i.e. for any given assignment to PV, only a single c_i evaluates to true, and e_i will be executed if c_i evaluates to true.*
- *$REPEAT$ represents a loop structure and is defined as a tuple $(pe, c\{pe_i\})$, where c is a logical condition as already defined, and $pe, pe_i \in E$. c is evaluated just after the end of pe, and if it holds, then pe is repeated, after the execution of the optional pe_i.*
- *$WHILE$ is similar to $REPEAT$, with c being evaluated before pe starts.*
- *DS is a dependency scope as defined in Definition 6.2.7.*

Definition 6.2.6 (Activity (ACT))**.** *Given a service repository SR, an activity is a process element E which represents one of the following constructs:*

- *the invocation of a service instance, with $act = (id(act),\ in(act), out(act))$, where:*
  - *$id(act)$ is a reference $stid.iid.oid$, with $stid$ being an identifier of a service type $st = (stid, O, SV) \in ST$, iid the identifier of a service instance si =$(iid, stid) \in SI$, and oid is the identifier of some operation $o \in O$.*
  - *$in(act) = in(oid)$.*
  - *$out(act) = out(oid)$.*

In BPEL it may correspond to an $invoke$, $receive$, $reply$, etc.

- *the idle activity $no\text{-}op$, which corresponds to $empty$ in BPEL.*
- *the special activity $exit$, whose execution causes the entire BP to halt (corresponding to $exit$ in BPEL).*

The input (output) parameters of all activities in the BP form the sets IP (OP). Input variables can be assigned with constant values or other process variables: $id(act)(ip_1 := v_1, \ldots, ip_n := v_n)$, where $ip_i \in in(act)$, $v_i \in PV$, or v_i is a value compliant with ip_i's domain. The activity outputs can be stored in some local process variable: $pv_i := op_i$, where $op_i \in out(a)$ and $pv \in PV_i$.

6.2.2 Dependency scope

The DS is a *guard-verify* structure, where the critical part of the BP is included in the *guard* block, while the *verify* block specifies the types of events that require intervention. Whenever such an event occurs, the control flow is transferred to the *verify* block, and the respective goal is activated. Once the resulting IP finishes execution in the updated environment, the control flow of the BP continues from the point following the *guard-verify* structure, unless it is explicitly forced to terminate.

Definition 6.2.7 (Dependency Scope (DS))**.** *Given a service repository SR and a business process $BP = (PV_i \cup PV_e, E)$, a dependency scope is a tuple $DS = \langle guard(VV)\{CS\}, verify(\{(case(C_i)\colon G_i \mid E_{ip} \mid terminate(G_i) \mid terminate(E_{ip}))\})\rangle$, where:*

- *$guard(VV)$ indicates the set of volatile variables $VV \subset PV_e$ whose modification triggers the verification of the DS, and $CS \in E$ a process element, which is called the* Critical Section*. Whenever during the execution of CS an event indicating a change in the value of a volatile variable $vv \in VV$ is received, the $verify$ part of the DS is triggered, and the execution of BP is interrupted.*
- *$verify(\{(case(c_i) : G_i \mid E_{ip})\})$ comprises a set of tuples consisting of a case-condition c_i and a goal G_i or a process element E_{ip} to be pursued if c_i holds.*

  - *c_i is a logical condition C. Providing a case condition is optional, with the default interpretation being $c_i = TRUE$.*
  - *G_i specifies a goal, which ensures the satisfaction of the properties that reflect the state right after the final activity of E_g. G_i is specified in the goal language supported by the planner as presented in (Kaldeli et al., 2009). After interrupting the BP execution, the plan that satisfies the respective G_i (if it can be found) is executed. When the execution of the plan is completed, the BP is resumed at the state after CS and from any other parallel branches of the BP that were interrupted.*
  - *If an E_{ip} is pre-specified to be executed in case C_i holds, then the execution of BP is interrupted, E_{ip} is executed, and after its completion BP resumes from the end of CS.*

- *$terminate(G_i)$ ($terminate(E_{ip})$) forces the process to terminate, i.e. abort the execution of BP, after fulfilling G_i (completing the execution of E_{ip}).*

Please note that the Critical Sections described here are not related to critical sections as defined in operating systems research (Dijkstra, 1965; Lamport, 1987), as the concurrent access to a certain data element is not and cannot be restricted to a single process. That is, access to the data element is not locked.

The complete specification of the full WMO process, annotated with all DSs, is provided in 8.8. Following Definition 6.2.7, the DS specification representing DS_1 of Figure 5.6 is the following:

```
<DS>
  <guard>
    <variables>
      <variable name="bpAddress"/>
      <variable name="bpMedCond"/>
    </variables>

    <!-- Subprocess covered by DS1 as in Figure 2 -->

  </guard>
  <verify>
    <case condition="bpAddress.county!='Groningen'">
      <terminate>
        <achieve-maint>
          <eq-val var="notifiedCityHall" value="TRUE"/>
          <eq-val var="messagePar" value="countyChange"/>
          <invalid var="orderId"/>
        </achieve-maint>
      </terminate>
    </case>
    <case condition="bpAddress.county='Groningen'␣AND␣bpMedCond!='deceased'">
      <achieve-maint>
        <known variable="dlOut_conf"/>
      </achieve-maint>
    </case>
    <case condition="bpMedCond='deceased'">
      <terminate>
        <achieve-maint>
          <invalid variable="orderId"/>
        </achieve-maint>
      </terminate>
    </case>
  </verify>
</DS>
```

According to DS_1, if a modification in the address or the medical condition occurs within the scope of the guarded subprocess, the following goals are pursued:

- If the address change indicates that the citizen has moved outside of the municipality, the goal ensures that the intervention plan leads to a state, where the order for a wheelchair or home modification (depending on the value of the “provision” variable, which is determined by the activity “Intake and Application”) has been cancelled, and a respective notification is sent to the city hall. The plan will be equivalent to IP (e) of Figure 5.7.
- If the new address of the customer is still within the range of the municipality or/and the medical condition has changed to some new value that does not indicate “deceased”, the final desired state is that the delivery of wheelchair or home modification is performed by taking into account the new situation (the new medical condition and/or address). Depending on the state at which the modification occurs and the kind of the modification, the generated plan is one of the IPs (a) to (d) of Figure 5.7. After the execution of the plan the BP

execution resumes to handle the invoice.

- If the new value of medical condition indicates “deceased”, then the goal specifies that the order should be invalidated.

Depending on the state of the DS in the original BP, at which the relevant volatile variable modification was identified, the generated plan may vary considerably for the same goal. This way, one DS definition covers all forms of IPs specified in Figure 5.7, which are generated automatically by the planner. The domain designer just prescribes in the goal what properties have to be satisfied during recovery, but is not required to know the combinations of actions that can achieve the goal. The planner uses a heuristic that promotes optimal plans. As a result, the planner may come up with different plans that fulfill the goal, depending on the available services. Considering, for example, an address change after an order has been sent in DS1 in Figure 5.6. If the supplier service offers an $updateOrder$ operation, the planner will advocate an update in the order address information, instead of cancelling the existing order and sending a new one.

Interdependencies between variables are also defined on top of the BP specification, prescribing the direct dependency of some variables on the validity of some other variable. The $dependsOn$ relation is used for this purpose and adheres to following syntax: $dependsOn(v) = \{v_1, \ldots, v_n\}$. Whenever a change in variable v is discovered or whenever v is invalidated (by transitivity, as an effect of some other variable interdependency) by the PE, the direct invalidation of the current values of $v_1, \ldots, v_n$ is automatically implied, without the need of some special-purpose process to take care of that. For example, *dependsOn(bpAddress_address)* = $\{$*hvOut_homeInfo*$\}$, since $hvOut\_homeInfo$ refers to the information retrieved for the specific $hvIn\_address$. Thus, if the person moves to some other address, the collected information is not valid anymore. In turn, a set of variables are directly dependent on $hvOut\_homeInfo$, like $arOut\_ requirements$ reflecting the acquired requirements concerning the wheelchair. On the other hand, an $orderId$ is not directly dependent on the address, since it remains valid after these variables change, unless some other course of interaction actively cancels it. These additional statements are of particular relevance when the change of a volatile variable is discovered, so that all information directly dependent on the consistency of the volatile variable also becomes obsolete, as shown in Section 6.3.3. The full set of variable interdependencies that accompany the WMO process specifications are provided in Appendix B.

6.2.3 The planning domain

Given a request by the Process Executor in case a DS is triggered, the Domain Generator constructs a planning domain given a BP specification and an SR, which is used by the planner for generating the IPs upon recovery requests. In the following, we provide the definition of a *Planning Domain (PD)*, in line with (Kaldeli et al., 2011) (the automatic composition of the PD is described in Section 6.3). The planning domain has some special characteristics that distinguish it from classical planning representations. The domain accommodates for numeric variables, which can range over finite domains, including the input arguments of actions. In addition, numeric functions and effects beyond mere assignments are supported. The planning domain is enriched with a knowledge-level representation to model observational actions, whose invocation provides some new information that is unknown offline. Observational actions model data-providing services, which constitute the largest proportion of nowaday's services. The knowledge-level representation allows us to address switches with conditions on the outcome of such actions, as will be explained below. A step-by-step explanation of the automatic composition of the PD is provided in Section 6.3.

Definition 6.2.8 (Planning Domain (PD))**.** *A Planning Domain is a tuple* $\mathcal{PD} = \langle Var, Par, A \rangle$*, where:*

- Var *is a set of variables. Each variable* $v \in Var$ *ranges over a finite domain* D^v*.*
- Par *is a set of variables that play the role of input parameters to members of* A*. Each variable* $p \in Par$ *ranges over a finite domain* D^p*.*
- A *is the set of actions. An action* $a \in A$ *is a tuple* $a = (id(a),\ in(a), precond(a),$ $effects(a))$*, where:*

- $id(a)$ *is a unique identifier*
- $in(a) \subset Par$ *are the input parameters of* a
- $precond(a)$ *is a propositional formula over* $Var \cup Par$*, which conforms to the following syntax:*
  $precond(a) ::= prop | \wedge_i\, precond(a) | \vee_i\, precond(a) | \neg precond(a)$
  $prop \quad ::= var \circ val \ | \ var1 \circ var2 \ | (var1 \diamond var2) \circ val \ |$
  $\qquad known(var),$

  where:

  * $var, var1, var2 \in (Var \cup Par)$.
  * val *is some constant.*
  * $\circ$ *is a relational operator* ($\circ \in \{=, <, >, \neq, \leqslant, \geqslant\}$).
  * $\diamond$ *is a binary operator* ($\diamond \in \{+, -\}$).
- $effect(a)$ *is a conjunction of any of the following elements:*
  * $assign(var, v)$*, where* v *is some constant or* $v \in Var$
  * $assign(var, f(v_1, v_2))$*, where* $v_1, v_2 \in (Var \cup Par)$ *or* v_1, v_2 *are constants, and* f *the sum or the subtract function*
  * $increase(var, v)$ *or* $decrease(var, v)$*, where* $v \in Var$ *or* v *is some constant*
  * $sense(var)$*, where* $var \in Var$.
  * $invalidate(var)$*, where* $var \in Var$*. This effect states that* var *becomes unknown.*
  * $prop(a) \Rightarrow effect(a)$*, which models a conditional effect.*

The output variables of an action are included as part of its sensing effects, i.e. they are assigned a value which is unknown offline, and can be any value that is consistent with the variable's domain. A state s is defined as a tuple $s = \langle (x_1, D_s^{x_1}), \ldots, (x_n, D_s^{x_n}) \rangle$, where $x_i \in Var \cup Par$ and $D_s^{x_i} \subseteq D^{x_i}$. The domain of x at state s is given by the *state-variable* function $x(s)$, so that $x(s) = D_s^x$ if $(x, D_s^x) \in s$. If $|D_s^x| = 1$, this means that x at s has a specific value. The domain modelling is based on the Multi-valued Planning Task encoding (Helmert, 2009), which leads to a smaller number of variables ranging over larger domains, and is particularly well-suited for constraint solvers. The effects of type $sense$ are called *observational*, i.e. they observe the current value of a variable, while the $assign$ and $increase/decrease$ types of effects are *world-altering*, i.e. actively change the value of a variable. An action may have both observational and world-altering effects.

Sensing effects are particularly important to model situations that involve non-deterministic assigments to variables. For example, the result of the *deferred choice* after the "Decision" action in Figure 3.6 (i.e. $dcOut\_confirm$=$true$ or $dcOut\_confirm$ = $false$) is modelled via an effect of the form $sense(dcOut\_confirm)$. Sensing outcomes are commonly used in deferred choices, (i.e. switch constructs) where the condition depends on some interaction with the operating environment. Its verification is thus deferred until runtime, after some variable is determined during the execution of a knowledge-providing action. The $invalidate$ type of effects indicate that the value of a variable is not valid, and should therefore not be used by subsequent actions before being observed again, in order to derive a sound value. For example, the action $cancelOrder(orderId)$ has as an $invalidate(orderId)$, which entails that the $orderId$ of an order that was processed is no longer valid.

Conditional effects can be used to model deferred choices, where different effects are materialized, depending on which proposition holds. For example, the negative effect of the activity "Check Tender with Decision" (if the tender selection is not approved by the municipality) entails the invalidation of the "Tender Procedure" outcome for selecting the company to undertake the home modification. As a result, the repetition of the "Tender Procedure" is enforced for the process to go on. This behaviour is modelled by the effect $\neg tsOut\_tenderSelOK \Rightarrow invalidate(ctOut\_tenderSel)$, and is automatically generated given the repeat structure of the BP specification, as it is explained in Section 6.3.

The domain is extended with additional variables to model the knowledge-level representation, and to distinguish between sensing and world-altering actions. These variables are generated automatically given a planning domain $\mathcal{PD}$. First, for each $var \in Var$, a new boolean variable var_known is introduced, indicating whether var is known at state s ($var\_known(s) = true$) or not ($var\_known(s) = false$). Given these additional knowledge-level variables, $known(var)$ is equivalent to $var\_known = TRUE$. Similarly, $invalidate(var)$ is equivalent to $assign(var\_known, FALSE)$. For every variable $kvar \in Var$ that participates in an observational effect, a new variable is introduced $kvar\_response$, which is a placeholder for the value returned by the respective sensing operation. Since this value is unknown until execution time, $kvar\_response$ ranges over $kvar$'s domain ($kvar\_response \in D^{kvar}$). Thus, $sense(kvar)$ is equivalent to $assign(kvar, kvar\_response)$. Furthermore, for each variable $cvar \in Var$ that is part of at least one world-altering effect, a boolean flag is maintained, which becomes true whenever this effect takes place. Consequently, the extended set of variables $V = Var \cup Par \cup Kb \cup Cv \cup Rv$ is obtained, where Kb is the set of knowledge-base variables, Cv the set of the change-indicative variables, and Rv the response variables.

6.2.4 Encoding the domain into a CSP

The PD can be mapped into a Constraint Satisfaction Problem (CSP), which can in turn be passed to a constraint solver, together with some goal that is expressed in the form of constraints (see (Kaldeli et al., 2011)). The computed solution to the CSP (assignment to variables) amounts to a plan (partially ordered sequence of actions) that satisfies all the constraints imposed by the domain and goal.

Formally, a constraint satisfaction problem is a triple $CSP = \langle X, \mathcal{D}, \mathcal{C} \rangle$, where $X = \{x_1, \ldots, x_n\}$ is a finite set of n variables, $\mathcal{D} = \{D^1, \ldots, D^n\}$ is the set of finite domains of the variables in X so that $x_i \in D^i$, and $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a finite set of constraints over the variables in X. A constraint c_i involving some subset of variables in X is a proposition that restricts the allowable values of its variables. A solution to a CSP $\langle X, \mathcal{D}, \mathcal{C} \rangle$ is an assignment of values to the variables in X $\{x_1 = v_1, \ldots, x_n = v_n\}$, with $v_i \in D^i$, that satisfies all constraints in $\mathcal{C}$.

Following a common practice in many planning approaches, we consider a *bounded* planning problem, i.e. we restrict our target to finding a plan of length at most k, for increasing values of k. Considering a planning domain extended with the knowledge-level variables $PD = \langle V, A \rangle$, the target is to encode PD into a $CSP = \langle X_{CSP}, \mathcal{D}, \mathcal{C} \rangle$. First, for each variable $x \in V$ ranging over D^x, and for each $0 \leqslant i \leqslant k$, we define a CSP variable $x[i]$ in CSP with domain D^x. Actions are also represented as variables: for each action $a \in A$ and for each $0 \leqslant i \leqslant k-1$ a boolean variable $a[i]$ is defined. The assignment to these action variables represents the plan. That is, an action is executed at state i if $a[i]$ is true. It should be noted that the computed plan may include parallel actions (since multiple action variables can be assigned to true at the same state). If some action a_1 affects a variable that is part of the preconditions of some other action a_2, or if both affect the same variable, then a_1 and a_2 are prevented from being put in parallel by an additional constraint.

Action preconditions and effects, as well as frame axioms, are automatically encoded as constraints on the CSP state variables, based on the formulation described in (Ghallab et al., 2004). Frame axiom constraints are also generated, which guarantee that variables cannot change between subsequent states, unless some action that affects them takes place. For every $v \in Var - (Par \cup Rv)$ and for for each $0 \leqslant i \leqslant k - 1$ the constraint $\wedge_j(actionAff(v)_j) = 0) \Rightarrow v[i] = v[i + 1]$ is added, where $actionAff(v)_j$ are the actions affecting v.

6.3 Automatic intervention process generation

In this section, the preliminary steps required for IP generation are explained. These steps comprise the formation of the atomic actions, the generation of a planning domain by the DG and the formation of the initial planning state by the PE. Furthermore, it is explained how complex BP-constructs are handled by the AI planner.

6.3.1 Formation of the atomic actions

The semantic specifications stored in the Service Repository are process-independent, and capture the generic functionality of the respective service operations in terms of preconditions and effects, so that they can be used in the context of various BPs. Usually these preconditions and effects concern the set of inputs and outputs of the respective operations and some additional aspects that are internal to the particular service.

For each BP, the operations of a subset of service instances in the Service Repository are marked as pertinent compensation methods. These methods can be part of the intervention processes for repairing the BP, and are annotated by the domain designer. If a permissive approach is adopted, the entire set of service instances in the SI part of the SR is allowed to be used by the IP. These compensation methods, along with the invocation methods referenced by the activities in the BP, form the *BP-Pertinent Methods (BPPM)* set. For each method $stid.iid.oid \in BPPM$ of a service instance $si = (iid, stid) \in SI$, whose service description includes an operation o with $id(o) = oid$, the PE generates some instance-level variables, preconditions, and effects, based on its iid and the operation description o this method realizes. The resulting set of instance-level method descriptions forms the *Atomic Actions*.

Atomic Actions (AA). Given a service repository SR, a business process BP, and a set of BP-pertinent methods *BPPM*, the Atomic Actions (AA) are formed as follows:

- When the PE receives a request to execute the BP (i.e. for every new process instance), a unique instance reference $bp\text{-}iid$ is assigned.
- For each method $bpo = stid.iid.oid \in BPPM$, the service type $st = (stid, O, SV) \in ST$ is found, and the operation $o = (id(o), in(o), out(o), prec(o), \mathit{eff}(o)) \in O$ with $id(o) = oid$ is retrieved.
- For each input parameter $ip_i \in in(o)$, a new input variable is created for $stid.iid.oid$, with name $bp\text{-}iid.stid.iid.oid.ip_i$ and a domain identical to ip_i. Similarly, for each output parameter $op_i \in out(o)$, a new output variable is created, with name $bp\text{-}iid.stid.iid.oid.op_i$ and a domain identical to op_i. The resulting instance-level input and output parameters form the sets $in(bpo)$ and $out(bpo)$ respectively.
- Based on the preconditions and effects of o, the sets $prec(bpo)$ and $\mathit{eff}(bpo)$ are generated, by substituting each input and output parameter with name v appearing in $prec(o)$ and $\mathit{eff}(o)$ by the reference $bp\text{-}iid.stid.iid.oid.v$. In case of a service state variable $var \in SV$ with local name v, the reference is substituted with the universal name $stid.iid.v$, which is BP independent. If $stid.iid.v$

has not been met before, the respective variable with name $stid.iid.v$ and with domain identical to var is created.

This way, for each $act = stid.iid.oid \in BPPM$ the invocation method description $imd = (bp\text{-}iid.stid.iid.oid,\ in(act),\ out(act),\ prec(act),\ eff(act))$ is created by the PE. Each imd is converted to a planning action (see in Definition 6.2.8) $a = (id(a) = (bp\text{-}iid.stid.iid.oid, in(a_i) = in(act)), prec(a) = prec(act), eff(a) = eff(act))$. These actions form the AA. The set of the instance-level inputs and outputs of all $bpo \in BPPM$ form the *Atomic Inputs* (AI) and the *Atomic Outputs* (AO) respectively, while the service state variables involved in the precon ditions or effects of the service types of all $bpo \in BPPM$ form the *Atomic Service Variables* (ASV).

The AA together with the set of variables AI, AO, ASV formed as described in the definition above, reflect only the atomic-level semantics of the actions. In the context of a certain BP, the universal action descriptions in the AA have to be enriched with extra preconditions and/or effects, which reflect the process-specific interdependencies, and which can be automatically inferred from the structure of the BP.

6.3.2 Generation of the planning domain

The Domain Generator (DG) is responsible for transforming the AA to a Planning Domain. A Planning Domain comprises a process-specific representation of actions participating in the particular BP (to restrict their use according to the BP structure) as well as the compensation activities that are allowed to be used by the respective IPs. The first time the PE needs to call the AI Planner at a certain process instance, the DG is called to generate the Planning Domain. Throughout the entire process instance, the same Planning Domain can be used. Consequently, the DG is only required once for a certain process instance. In the following, it is explained how these additional semantics are added to the atomic descriptions of the actions, in order to capture process-specific constraints.

Some additional assumptions regarding the BP definition given in Definition 6.2.5 have to be made, which allow us to derive all process-specific preconditions and effects in an automatic way from the BP specification. Given a repeat structure

$repeat = (pe, c\{pe_i\})$, if the optional intermediate pe_i is empty, it is assumed that in case c holds, the outcomes of the activities in pe are automatically invalidated, in order to to enforce the repetition of $firstAct(pe)$. For example, if the outcome of "Check tender with decision" is negative, another tender has to be selected. As a result, the output of "Tender Procedure" (the supplier selection) has to be invalidated.

On the other hand, in case a pe_i is intervened before pe, some activity in pe_i should take care of the invalidation of the relevant outcomes of the actions in pe (as e.g. is the case with "Return invoice to the supplier"). These additional restrictive assumptions are not necessary if the extra preconditions and effects are added explicitly by the domain designer.

Algorithm 1 takes as input the BP specification, and the set of atomic actions AA (which comprise the activities participating in the BP plus the allowed compensation actions). By parsing the BP, it constructs the appropriate preconditions and effects for each activity that is part of the BP. These preconditions and effects are added on top of the atomic functional preconditions and effects of the respective action in the AA. The BP is treated as a tree (represented as an XML tree), where the root is the outer-most element in the specification, and the leaves are the activities. For each element its parent can be obtained, and given an element one can reach its children. The parsing starts from the root and gets the next element in a depth-first way. If the element is an activity a, first its inputs are parsed: for each assignment to an input parameter, the respective equality proposition is added to the preconditions of a. Next, possible assignments of the outputs of a to BP variables of the form $bpVar := eOut\_v$ are parsed, and the respective $assign$ effect is added to the effects of a.

The preconditions enforcing the sequence relation of a with respect to its preceding process element e, as computed by the PREVELEM function in Algorithm 3, are returned by the function SEQPREC in Algorithm 3. These preconditions ensure that the appropriate preceding actions are executed prior to a, depending on the type of e. More specifically, SEQPREC obtains the preconditions corresponding to all execution paths that may lead to a, by finding the last action(s) of the respective execution

Algorithm 1 Automatic addition of BP-specific preconditions and effects given a BP specification and a set of atomic actions AA. The resulting set of BP-specific action descriptions constitutes the Planning Domain.

```
procedure PD($BP, AA$)
  while hasNext(BP) do
    $e$ = getNextElement($BP$)   //depth-first parsing of the BP tree
    match $type(e)$
      case $activity$:
        while hasNextInput($e$) do   //parse input assigments
          ($ip_i := v$) = parseNextInput($e$)
          addPrec(getAction($id(e), AA$), '$ip_i = v$')
        end while
        while hasNextOutAssign($e$) do   //parse possible assigns of outputs to vars
          ($bpVar := eOut\_v$) = parseNextOutAssign($e$)
          addEffect(getAction($id(e), AA$), '$assign(bpVar, eOut\_v)$')
        end while
        addPrec(getAction($id(e), AA$), SEQPREC(PREVELEM($e$), $BP$))
      case $switch\{(c_1, e_1), \ldots, (c_n, e_n)\}$:
        while hasNextBranch($e$) do   //parse all branches of the switch
          $(c_i, e_i) =$ getNextBranch($e$)   //precs for all actions at the beginning of swich
          $\forall a_i \in$ FIRSTACT($e_i$): addPrec(getAction($id(a_i), AA$), '$c_i$')
        end while
      case $repeat(pe, c)$:   //$e$ is a repeat without an intermediate $pe_i$
        $\forall a_i \in$ LASTACT($pe$):   //effects for all actions after the loop $pe$:
          //invalidate the outputs of all actions in the repeat loop
          addEffect(getAction(id($a_i$), AA), '$c \Rightarrow \wedge_{a_j \in pe, o_k \in out(a_j)} invalidate(o_k)$')
      case otherwise: continue
  end while
end procedure
```

paths, and the possible respective conditions on which this path is depending.

The function PREVELEM(a, BP) returns either the previous element of a in a sequence relation if such one exists, or otherwise it recursively returns to the ancestors of a, until it reaches a sequence relation. If no sequence exists in its roots, there is no activity preceding a. If e=PREVELEM(a, BP) is an activity, the precondition states that the outputs of e have to be known. If e is a sequence, then SEQPREC is computed on the last element in that sequence. In case of a repeat-construct, SEQPREC is called recursively on the loop element. Moreover, the negation of the condition at the end of the loop should hold for the control flow to proceed with the execution of a.

Algorithm 2 Function for computing preconditions capturing sequence relations. The computed preconditions are added to the action that follows in the BP.

```
function SEQPREC($e, BP$): Precondition
  match type(e)
    case $activity$:
      return '$\wedge_{o_j \in out(e)}$ $known(o_j)$'   //action's outputs are valid
    case $seq\{e_1, \ldots, e_n\}$:  SEQPREC($e_n, BP$)
    case $repeat\{pe, c\{e_i\}\}$:  return $\neg c \wedge$ SEQPREC($pe, BP$)
    case $switch\{(c_1, e_1), \ldots, (c_n, e_n)\}$:     //$e_i$ of switch-branch $c_i$ is valid if $c_i$
      return '$\wedge_i$ ($\neg c_i \vee$ SEQPREC($e_i, BP$))'
    case $flow\{e_1, \ldots, e_n\}$:     //all parallel $e_i$s are valid
      return '$\wedge_i$ SEQPREC($e_i$)'
    case $empty$:
      if PREVACT($e$) $\neq \varnothing$ then
        SEQPREC(PREVACT($e, BP$))
      else
        return true
      end if
end function
```

For multiple incoming branches in the case of flow, the sequence preconditions modelling all elements in the flow are obtained. If the e is of type $switch = \{(c_1, e_1), \ldots, (c_n, e_n))$, the preconditions state that the element e_i should be executed prior to a only *if* the respective branch was taken, i.e. if condition c_i holds. Finally, if e (the previous element with respect to the parent element of a) is the $empty$ activity, and $parent(a)$ is not the root of the BP, then the algorithm proceeds recursively in computing the sequence preconditions entailed by the ancestors of e.

Algorithm 3 Auxiliary function for obtaining the previous element in a sequence.

```
function PREVELEM($e, BP$): Element   //Returns the previous element of $e$
  match type(parent($e, BP$))
    case $seq\{e_1, \ldots, e_n\}$:
      if $e = e_i \wedge i \neq 1$ then
        return $e_{i-1}$   //if $e = e_i$ not last in seq, return $e_{i-1}$
      else   //if last, the previous is the previous of the parent
        PREVELEM(parent($e, BP$))
      end if
    case otherwise:
      if parent($e, BP$)=$\varnothing$ then   //if root
        return $\varnothing$
      else   //in all other cases, previous is the previous of the parent
        PREVELEM(parent($e, BP$))
      end if
end function
```

After taking care of the sequence preconditions, Algorithm 1 proceeds with checking the case where the current element in the tree is of type switch. In this situation, for each branch(c_i, e_i) of the switch the condition c_i is added as a precondition to the first activity(ies) of e_i. These first activities are computed by the function FIRSTACT in Algorithm 4. FIRSTACT recursively obtains the first element(s) of e_i, depending on the type of e_i, until this element is an activity. In the next step, if e = $repeat(pe, c)$, a conditional effect is added, which invalidates the results of all actions in the loop element pe, in case the repeat condition c holds, in order to compel their repetition. In Section 8.8 the final planning domain representing the WMO process, as produced by Algorithm 1, is presented.

The outcome of the algorithm is a *BP-specific Actions Set (BPAS)*, which is the original AA enriched with the extra preconditions and effects. Together with the set of variables consisting of the variables AI, AO, ASV as described in Section 6.3.1 and the internal process variables PV_i declared in the BP, they constitute the planning domain considered by the planner. The BP-specific planning domain is thus defined as $PD = \langle Var, Par, Act \rangle$ (see Definition 6.2.8), with $Var = PV_i \cup AO \cup ASV$, $Par = AI$, and $Act = BPAS$.

6.3.3 Formation of the initial planning state

The initial planning state comprises the values of all variables at the current state of execution and the knowledge level with respect to the variables interdependency rules. Given the manually specified variable interdependencies in terms of the $dependsOn$ sets, these are enriched during execution of the BP by the PE: if an action comprising an assignment effect $assign(v', v)$ or an increase(decrease) effect $increase(v', v)$ ($decrease(v', v)$), has been executed, variable v' is added automatically to the $dependsOn(v)$ set (if the set does not already exist, it is created). Each time the AI planner is called by the PE, the initial planning state is formulated as follows:

- Each variable $var \in PV$ is equal to a value corresponding to the state of execution, i.e. considering the assignments to the BP input parameters, the outputs of the service invocations, the assignments to variables, and the received external events (for more details see Section 7.1).

- For each variable var for which no specific value has been acquired yet, the respective knowledge variable $known\_var$ is set to false at the initial state ($known\_var(0) = false$).

- Given a change event on a volatile variable vv, the interdependency rules are parsed. For each $var \in dependsOn(vv)$, $known\_var(0) = false$, indicating that the value of var as reflected by the current state of execution is not valid. The same is done recursively for each $var' \in dependsOn(var)$, for all $var \in dependsOn(vv)$.

Algorithm 4 Auxiliary functions used for adding switch and repeat conditions as preconditions.

```
function FIRSTACT($e, BP$): Set[Element]    //Find the first action(s) of an element
  match type(e)
    case $switch = \{(c_1, e_1), \ldots, (c_n, e_n)\}$:
      return FIRSTACT($e_1, BP$) $\cup \ldots \cup$ FIRSTACT($e_n, BP$)
    case $repeat = \{pe, c\{pe_i\}\}$: return FIRSTACT($pe, BP$)
    case $flow\{e_1, \ldots, e_n)\}$:
      return FIRSTACT($e_1, BP$) $\cup \ldots \cup$ FIRSTACT($e_n, BP$)
    case $seq\{e_1, \ldots, e_n\}$: return FIRSTACT($e_1, BP$)
    case $activity$: return e
end function

function LASTACT($e, BP$): Set[Element]    //Find the last action(s) of an element
  match type(e)
    case $switch = \{(c_1, e_1), \ldots, (c_n, e_n)\}$:
      return LASTACT($e_1, BP$) $\cup \ldots \cup$ LASTACT($e_n, BP$)
    case $repeat = \{pe, c\{pe_i\}\}$: return LASTACT($pe, BP$)
    case $flow\{e_1, \ldots, e_n)\}$:
      return LASTACT($e_1, BP$) $\cup \ldots \cup$ LASTACT($e_n, BP$)
    case $seq\{e_1, \ldots, e_n\}$: return LASTACT($e_n, BP$)
    case $activity$: return e
end function
```

6.3.4 Generating the intervention process

By starting from the initial state as delivered by the PE, and depending on the goal, the IP can be computed by the AI planner using the planning domain. This IP may include the re-invocation of activities with the up-to-date input parameters, if this is required to achieve the goal (e.g. pay a visit to the new address to acquire the informed requirements), or try to find a sequence of “undo” actions that actively lead to the invalidation of some variables (e.g. try to cancel an order that has been sent if possible).

In case of deferred choices (i.e. XOR-constructs (switches) where the value of a variable participating in the respective condition is unknown off-line) it has to be ensured that the right branch is followed at runtime. One way to address this issue is to rely on conditional plans, as e.g. presented in (Pistore et al., 2005; Hoffmann et al., 2012). However, for these approaches it is difficult to deal with sensing outcomes that range over numeric-valued domains. Herein, we resort to a re-planning mechanism to model deferred choices, where the value of the condition is acquired during runtime.

The plan originally returned by the planner is optimistic, i.e. the variables that are unknown off-line are assumed to have values that lead to the shortest plan that fulfills the goal. Thus, in the case of the IP Figure 5.7c, it generates the plan that corresponds to the assumption that the output of “HomeVisit” $hvOut\_maRequired = false$, which indicates that the home inspection does not entail the need for a medical advice, that the decision is positive, and that the supplier selected by the customer is approved. Whenever a knowledge-providing activity is executed by the PE, and the initially unknown variable is instantiated, the outcome is compared with the value assumed by the plan. That is, it is checked whether the new knowledge incorporated in the CSP violates any constraint. If no violation is detected, then the execution of the IP may proceed according to the initial plan. In case of a violation, the planner is invoked again with the same goal and a new initial state, including the value of the sensed variable. As a result, a request for a Home Modification may require the following series of interactions when planning for Goal *achieve-maint(known(delOut_delId))* (see Section 6.2.2), in order to obtain the IP shown in Figure 5.7c (the input parameters are omitted for brevity):

Initial plan: $\{HomeVisit, Decision, TenderProcedure, CheckTender, SendOrder, Delivery\}$

Execute $HomeVisit$ Output: $hvOut\_maRequired = true$, **constraint violation, re-plan**

New plan: $\{MedicalAdvice, Decision, TenderProcedure, CheckTender, SendOrder, Delivery\}$

Execute: $MedicalAdvice$ $maOut\_medInfo = `Document12A'$

Execute $Decision$ Output: $dcOut\_approvalCheck = true$

Execute $TenderProcedure$ Output: $tpOut\_tenderSelection = `ACMFrizianConstructions'$

Execute $CheckTender$ Output: $ctOut\_tenderOK = false$, **constraint violation, re-plan**

New plan: $\{TenderProcedure, CheckTender, SendOrder, Delivery\}$

Execute $TenderProcedure$ Output: $tpOut\_tenderSelection =$'van der Meer Elevators'
Execute $CheckTender$ Output $ctOut\_tenderOK = false$
Execute $SendOrderToSelSupplier$ Output: $soOut\_orderId = `14578AS'$
Execute $Delivery$ Output: $dlOut\_conf = `Delivered'$

If the output of "Decision" is negative, then no plan exists that satisfies the goal. In that case, the planner returns a message indicating that the goal is not satisfiable, causing the BP execution to be aborted. In total 9 service operations are invoked as part of the IP.

The IP generated by the planner is finite in all cases. Although the AI Planner can model finite loops (i.e. a repetition of certain activities), the IP cannot have indefinite loops, since the plan is a finite, partially ordered set of actions. The loops as a result of deferred choices are caused by BP specific preconditions and effects. The planner may be called indefinitely as a result of deferred choices, if the output of the sensing actions keeps satisfying the loop condition. To avoid such situations, an upper limit is put to the number of times the replanning process can be invoked.

6.4 Automatic identification of critical sections

The algorithm of automated generation of the parts of a BP covered by a DS is presented in Algorithm 5 below. The algorithm guarantees that the computed CSs are elements of the BP in compliance with Definition 6.2.5. CSs cover all activities that are directly or indirectly dependent on the same set of volatile variables VV. That is, they either use a $vv \in VV$ as input or use the output of another activity, which is dependent on vv. These activities are referred to as *Dependent Activities* (DA). In order to ensure that important change events will not pass untreated, any part of the process in a potential execution path between two activities dependent on the same VV should also be covered by the respective CS. This is necessary to take care of any modification of vv that occurs during the execution of this intermediate part, since the modification may require the cancelation or repetition of some preceding part of the BP which relied on some $vv \in VV$ (e.g. performing a new visit to the new house if the address has changed), and which is used by a succeeding

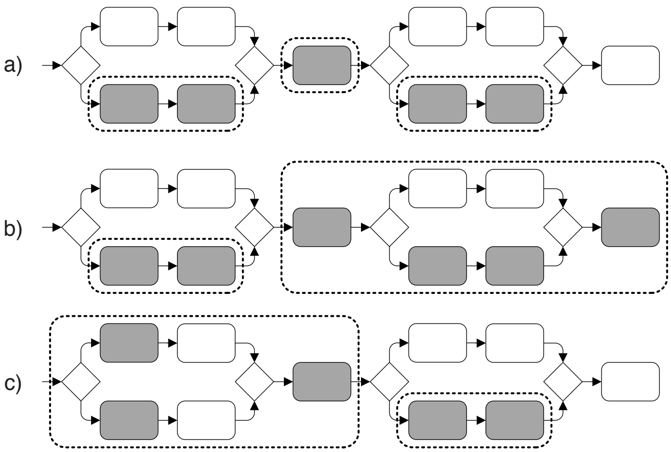


Figure 6.2: CS creation examples

element (e.g. to calculate the characteristics of the requested wheelchair). However, branches in switch or flow constructs that are not on a potential path between two activities dependent on some vv, should not be unnecessarily included in the respective CS, in order to avoid unnecessary invocation of intervention processes.

In Figure 6.2, some examples of CSs are provided to illustrate the properties described above. The shaded activities are dependent on VV and should be covered by a CS. The CSs are indicated by a dashed line. In case (a), only the specific branches of the switch-constructs that comprise dependent activities are included in the CS. In situation (b), however, the second switch has to be covered entirely by a CS, because the last activity is dependent on VV as well. Any modification event regarding a $vv \in VV$ that occurs during the upper branch (which is not dependent on VV) has still to be dealt with, since the last activity may use a a variable that is a result of some dependent activities before the switch, which produced this result based on the obsolete vv. In situation (c), both branches of the first switch contain activities that are not dependent on VV. However, as they both are on a path between activities that are dependent on VV, the entire switch is covered by a CS.

The main function of Algorithm 5 is *extractScopes*, which takes as an input a BP specification in accordance with Definition 6.2.5 and the list of volatile variables VV. *extractScopes* returns a list of tuples $\langle VV_i, CS_i \rangle$, which correspond to the

$guard$ parts of all DSs in the BP. Given a $BP = (PV_i \cup PV_e, E)$, $VV = PV_e$. That is, all state variables that are declared in the SR and used in the BP should be guarded, since their modification may be a source of erroneous results. The BP is treated as a tree (represented in XML), where the root is the outermost element in the specification, and the leaves are the activities.

Algorithm 5 Automatic computation of the set of the pairs Guard=$\{\langle VV_i, CS_i\rangle\}$, consisting of volatile variables and respective elements that constitute the Critical Sections

```
 1: function EXTRACTSCOPES($BP$, $VV$): List[(List[V], E)]
 2:   for each $vv \in VV$ do
 3:     $guardList = \emptyset$
 4:     $DE =$ GETDEPENDENTELEMS($vv, BP$)
 5:     for each $e_i \in DE$ do
 6:       $tmpCS = \emptyset$
 7:       $DE = DE$.remove($e_i$)
 8:       for each $e_j \in DE$ do
 9:         if type(minCommonAncestor($e_i, e_j$))=sequence then
10:           $tmpCS = tmpCS \cup$ GETTEMPCS($e_i$, $e_j$, $BP$)
11:           $DE = DE$.remove($e_j$)
12:         end if
13:       end for
14:       for $tmpCS_i \in tmpCS$ do
15:         $guardList$.add($\langle\{vv\}, tmpCS_i\rangle$)
16:       end for
17:     end for
18:   end for
19:   MERGESCOPES ($guardList$)
20: end function
```

The outermost loop in the function *extractScopes* iterates over the list of volatile variables VV. For each $vv \in VV$, critical sections are extracted separately. At the end, identical CSs for different variables are merged by $mergeScopes$ into a united CS. The first step (line 4) is to find all activities and switch–blocks that depend directly or indirectly on the volatile variable vv, by calling the function *getDependentElems*. First (line 24), all activities for which vv is assigned to some of their input parameters directly or by transitivity are added to the dependent elements DE. Then (line 34), DE is augmented by adding all switch–blocks whose condition is either on vv, or some variable produced by the already considered activities. All elements in DE are arranged in a breadth-first order as they appear in the BP.

Algorithm 6 Find all activities and switchblocks that depend directly or indirectly on the volatile variable vv

```
21: function GETDEPENDENTELEMS($vv, BP$): List[Element]
22:   $varList = \{vv\}$
23:   $DE = \emptyset$
24:   for each $a_i \in BP$.getActivities do
25:     for each $ip_i := v \in a_i$.parseInputAssignments do
26:       if $v \in varList$ then
27:         for each $op_i \in out(a_i)$ do
28:           $varList.add(op_i)$
29:         end for
30:         $DE$.add($a_i$); break;
31:       end if
32:     end for
33:   end for
34:   for each $SWITCH_i \in BP$.getSWITCHelements do
35:     $c_i = SWITCH_i$.getFirstCondition
36:     if $c_i.getLeftVariable \in varList$ then
37:       miDE.add($SWITCH_i$);
38:     end if
39:   end for
40:   return $DE$
41: end function
```

The next step in *extractScopes* is to iterate through the list DE. In the inner loop, for each pair of elements e_i, e_j, it is checked whether their minimal common ancestor is of type sequence. If so, then the function $getTempCS$ is called, which returns a set of elements that are candidates for being CSs with respect to the variable vv, and lie between e_i and e_j. Then, e_j can be removed from DE, since subsequent inspections on it are redundant, as the appropriate CSs covering it have already been computed.

Function *getTempCS(*e_i, e_j, BP*)* first calls *getPathBtw* to compute the path between e_i and e_j (line 44), which comprises all elements that are part of the sequence between e_i and e_j, including the special markers $StartBranchEl$ and $EndBranchEl$. These markers indicate the start (splits) and end points (joins) of branching elements. Consequently, a $path$ is a list with members of type $Item$ (line 59), where an item is either a process element or a $BranchElMarker$. Markers are added in the path only if they concern joins (splits) for which the respective split (join) is not encountered during the traversal of the BP from e_i to e_j. This way, the markers divide the path into the appropriate sequences of elements (lines 46 to 53), each of which is a candidate for being a CS.

Algorithm 7 Obtain temporary Critical Sections

```
42: function GETTEMPCS($e_i, e_j, BP$): List[Elem]
43:   $tmpCSList = \emptyset$
44:   $path$ = GETPATHBTW($e_i, e_j, BP$)
45:   $currCS = \varnothing$
46:   for each $item \in path$ do
47:     match type($item$)
48:       case Element:
49:         $currCS$.attachInSeq($item$)
50:       case BranchElMarker:
51:         $tmpCSList$.add($currCS$)
52:         $currCS = \emptyset$
53:   end for
54:   return $tmpCSList$
55: end function
```

Function *getPathBtw* uses the auxiliary function $nextItems$ (not explained in the algorithm for space reasons), which returns a list consisting of the next element in the sequence path, and some possible $EndBranchEl$, if any are encountered before the next element is fetched. These are added to the path, and the process proceeds by fetching the next items (line 60), until the element in the sequence that contains e_j is reached. In the latter case, *pathInElem* is called, which traverses the path within this last element until e_j is reached. If the element containing e_j is an activity or sequence, this activity (e_j) or the subsequence till e_j (line 70) are returned respectively. If the element is a switch or flow, then a *StartBranchEl* marker is added in the list of results, and the branch containing e_j is inspected. *pathInElem* is called recursively on this branch, and all items in the path leading to e_j are collected in $path_j$. Consequently, the computation of the entire path is completed, and returned to $getTempCS$. The path is traversed (line 46), and divided into the appropriate CSs: $currCS$ is constructed as a sequence of the elements in $path$, until a marker is met, at which point $currCS$ is added to the list of candidate CSs.

Once the list of temporary CSs $tmpCS$ regarding a volatile variable vv is computed as described above, $extractScopes$ proceeds with constructing the respective $guardList$ consisting of tuples $\langle \{vv\}, tmpCS_i \rangle$ (line 14). After repeating the process described above for each $vv \in VV$, $mergeScopes$ is called, in order to clean up the candidate CSs. The following steps are performed in that order:

- If there are two tuples $\langle \{v_1\}, CS_1 \rangle$ and $\langle \{v_2\}, CS_2 \rangle$, where CS_1 and CS_2 are identical, then they are replaced by a single tuple $\langle \{v_1, v_2\}, CS_1 \rangle$.

- If there are two tuples $\langle \{v_1\}, CS_1 \rangle$ and $\langle \{v_2\}, CS_2 \rangle$, where $v_1 = v_2$ and $CS_1.descendantOf(CS_2)$, then the former tuple is removed as redundant.

- If a list of tuples on the same volatile variable set $\langle VV, CS_1 \rangle, \ldots, \langle VV, CS_n \rangle$ correspond to the branches of a switch, i.e. there is an $e_{switch} = switch\{ (CS_1, e_1), \ldots, (CS_n, e_n)\}$, then these are replaced with a single CS, which covers the entire switch–element. A similar process is performed for flow branches.

- If a list of tuples on the same volatile variable set $\langle VV, CS_1 \rangle, \ldots, \langle VV, CS_n \rangle$ are interrelated through a sequence relation, i.e. there is a $seq\{CS_1, \ldots, CS_n\}$, then these are replaced with a single CS, which covers the entire sequence.

Algorithm 5 has been applied to the BP specification of the WMO process represented in Figure 3.6. The algorithm identified three volatile variables, and all five critical sections related to them. The total time for parsing the WMO process specification and computing all CSs is below 100 msec. The discovered CSs can then be projected on the Process Modeller, as presented in Chapter 7.

Algorithm 8 Function for computing elements that are candidates for critical sections.

```
function GETPATHBTW($e_i, e_j, BP$): List[Item]
  $currElem = e_i$
  while $\neg$ $currElem$.contains($e_j$) do
    $path$.append($currItems$)
    $currItems$ = NEXTITEM($currElem, e_i, BP$)
    $currElem = currItems$.getElement
    if $currItems = \emptyset$ then return $\emptyset$
    end if
  end while
  $path$.append(PATHINELEM($currElem, e_j, BP$))
  return $path$
end function

function PATHINELEM($el, endEl, BP$): List[Item]
  match type($el$)
    case activity:
      return $\{el\}$
    case sequence:
      return $el.subsequenceTill(endEl)$
    case SWITCH $\vee$ flow:
      $path_j = \{StartBrEl\}$
      $branch_j = el.getBranchWith(endEl)$
      return $path_j$.append(PATHINELEM($branch_j, endEl, BP$)
  return $\emptyset$
end function
```

CHAPTER 7

Implementation and evaluation

7.1 The prototype

The proposed approach for automatic process recovery upon data changes has been implemented in a prototype, comprising the components of the architecture outlined in Figure 6.1.

7.1.1 The process modeller

The Process Modeller (PM) is implemented in Java, by the use of standard Java 2D graphical libraries. It supports all basic BP modelling constructs, including SEQUENCE, FLOW, SWITCH etc., with an added support for DS modelling and generation. Furthermore, the PM provides for the declaration of the process variables, i.e. the definition of their name and type. However, the actual object creation is handled by the PE, which keeps and manages a local database as described in Section 7.1.2. The PM is connected to the Service Repository, so that the BP designer can use service operations that exist in the SR as activities in the BP being modelled.

Figure 7.1 presents a screenshot of the PM, showing the graphical representation

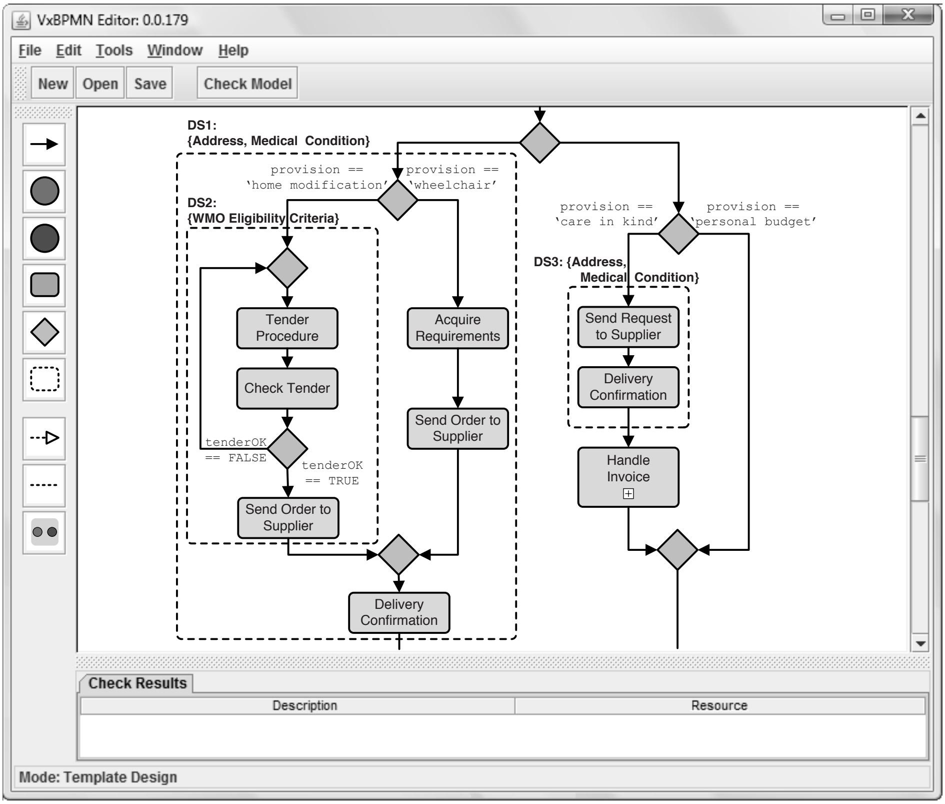


Figure 7.1: Screenshot of the Process Modeller.

of the DSs of the WMO process from Figure 5.6. The DSs are saved along with the rest of the process specification. The final output of the PM is an XML representation of the BP, which conforms to Definition 6.2.5. This representation is passed to the PE for execution, as described in the next subsection.

7.1.2 The process executor

The Process Executor (PE) is responsible for executing a BP as specified by the PM. The PE takes as an input a BP specification in conformance with an XML schema that represents Definition 6.2.5, and with the BP input parameters instantiated to specific values. The PE works in cooperation with the Service Repository as described in Definition 6.2.3. The details of the Service Instances implementation are outside the scope of this thesis. For testing purposes (presented in Section 7.2), the service invocations are simulated.

The activities included in the BP specification must refer to method invocations that can be retrieved from the SR. Given a fully qualified reference to an invocation method $stid.iid.oid$ specified by an activity in the BP specification, the PE retrieves the respective description kept in the SR.

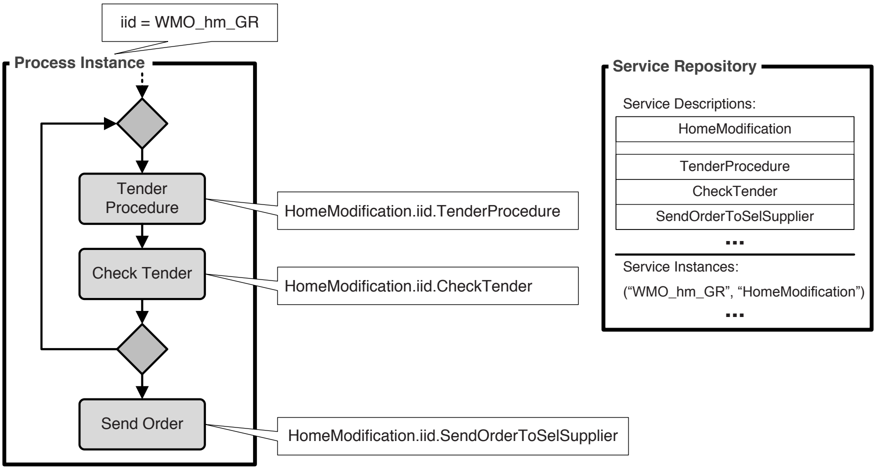


Figure 7.2: Example of a Service Type and a Service Instance.

For example, the activity "*Send Order*" in Figure 7.2 refers to "$HomeModification.iid.sendOrderToSel-Supplier$", which corresponds to the method "*sendOrderToSelSupplier*" of the "*HomeModification*" service type, and is provided by the service instance with identifier "WMO_hm_GR" (see Definition 6.2.3). As shown in Figure 7.2, the service type of "*HomeModification*" as well as the service instance (provider) "WMO_hm_GR" are kept in the SR. It should be noted that the value of the variable iid in the BP specification may be unknown before a process is actually started, and an assignment to another value $iid = iv$ can be used instead of a predefined value. The value of iv can be provided by the user at execution time, or retrieved by the PE as an output value of a service method call. In the example in Figure 7.2 the value "WMO_hm_GR" for the variable iid is provided at the time the process instance execution starts.

In the current implementation, an activity is executed by directly invoking the respective method, without checking whether the preconditions prescribed in the corresponding service instance description in the AA hold. Control flows are treated as by a typical execution engine.

The data flow and knowledge about the environment are handled by a *local storage* (LS), which is maintained by the PE and reflects its knowledge about the environment and the state of the process instance execution. Some of these variables are specific to a particular BP running instance, and some are common to multiple BPs. During execution, the PE updates the LS according to the new information it receives from the environment (from service method invocations), and to the specifications included in the BP description (assignments to variables). When the PE receives a request for executing an instance of a BP specification $BP = (PV, E)$, it assigns a unique identifier $bp\text{-}iid$ to the running instance, and constructs the AA along with the instance-level inputs and outputs $AI \cup AO$ (as described in Section 6.3.1), which are added to the LS. Each service state variable $sv \in ASV$ (see Section 6.3.1) is added to the LS if it does not already exist. This way, state variables of the AA are shared among running process instances, whereas instance-level input and output variables are unique to each process instance. Moreover, the PE constructs the instance-level internal variables declared in the BP (i.e. for each $var \in PV_i)$ with name v a variable with name $bp\text{-}iid.v$ and domain identical to var's domain is added to the LS. The internal process variables are also unique to the process instance. The value of an instance-level variable cannot be changed by any other external factor other than the BP instance $bp\text{-}iid$ it belongs to, while a shared variable can be modified by any other entity that calls the service operation which affects it.

The distinguishing feature of the PE with respect to other well-known BP execution engines is the support for dealing with the DSs specified in a BP. When a process execution runs into a DS, the PE turns into a special “DS mode”. In that mode, the PE creates an event listener for each of the volatile variables specified in the DS. It is assumed that modification events can be captured by subscribing to specific variables of interest, and that external services that have the permission to change these variables, publish an appropriate event that is caught by the subscribed clients (listeners). The details of event firing and catching are out of scope of the paper.

The event handling is deferred until the activity currently being executed finishes, thus avoiding potential inconsistencies that may result from canceling an activity in the middle of execution. Therefore, the information conveyed by the data modification events is stored in a memory list that maintains tuples of the recently modified variables and their latest values. A new event on the same variable overwrites the old value of the variable kept in the memory list. This list of recent changes is checked prior to executing the next activity within a DS, and if it is not empty, the conditions in the $verify$ block of the DS are checked towards the latest values kept in the list. If a condition evaluates to true, the respective goal or process element is fired, while the BP execution is suspended. In case of a flow, all parallel branches are put on hold. The list of recent changes is cleared, and the LS is updated accordingly, by incorporating the most up-to-date values to the respective variables.

In case a goal has to be pursued, the planner is invoked in order to create a plan which is then executed, while in the case of a pre-specified element this is directly executed. After a plan or a pre-specified element is executed the initial process execution is resumed, starting from the activity which is immediately after the end of the current DS. In case parallel branches were suspended, these are resumed as well (the underlying assumption is that the execution of the generated IP does not introduce any inconsistencies in the suspended concurrent branches). The only exception is when there is a $terminate$ annotation referring to the goal that is triggered (see Definition 6.2.7), in which case the original BP is terminated instead.

In case of nested DSs, the conditions are verified for all active dependency scopes starting from the most outer one and going inward. When the execution of a subprocess covered by some DS is finished, then the respective DS is removed from the list of active DSs, as well as all event listeners associated with it. If the list is empty, then PE leaves the “DS mode” and does not listen to any data modification events. Note that while executing an IP, the PE still remains in the same “DS mode”. The modification events received during the IP execution are treated in the same way as the execution of the process element covered by the DS in the BP. More specifically, an IP “inherits” the DSs that covered the activity responsible for invocation of the planner. In case a DS condition is triggered, the current IP execution is interrupted and new IP is generated. After the execution of that IP, the PE returns to the state after the DS in the original BP.

In order to generate a plan, the AI planner needs a planning domain representation (see Definition 6.2.8). To this end, the PE calls the Domain Generator, by passing to it the Atomic Actions (AA), built as described in Section 6.3.1 by including all service instances referenced in the BP and a set of eligible compensation services from the SR. The planning domain is constructed only once for a specific BP, the first time that a DS is triggered. The goal taken from the DS specification and the current state, i.e. the values of the variables that are part of the planning domain as reflected by the updated database, are handed over to the AI planner, which uses them along with the planning domain to compute a plan. This plan, which includes only sequence and flow structures, is then passed for execution to the PE. Loops in the plan are “flattened”, i.e. the plans explicitly include all repetitions in sequence. Deferred choices (such as in the case of switches) are addressed indirectly as already described in Section 6.3.4: whenever the PE executes an operation that returns a new value, the constraint solver is called to check whether this value leads to any inconsistencies with respect to the outcome anticipated by the plan. If that is the case, the planner is re-invoked with the current state of execution as the initial state (having the same goal).

7.1.3 The planner

The planner is implemented in Java, and communicates with the PE through standard method calls. Upon receiving a request for computing a plan from the PE, the planner translates the BP-specific planning domain, the initial state and the goal it received into a CSP, as presented in Section 6.2.4. A standard constraint solver is applied to solve the CSP, in order to find a solution that amounts to a valid plan. The Choco v2.1.1 constraint programming library[1] is used, which provides a large choice of implemented constraints, as well as a variety of pre-defined and custom

[1] www.emn.fr/z-info/choco-solver

search methods. The solution to a CSP amounts to a partially ordered plan, i.e. one that may contain parallel actions if not restricted by interdependencies between actions. This plan is passed to the PE for execution, as described in the previous section.

7.2 Evaluation

The aim of the evaluation is (i) to demonstrate the effectiveness of our approach with respect to our working example presented in Section 7.1 and (ii) to test the performance with respect to the time that is required to generate the necessary IPs. The specification of the desired goals and DSs has been conducted in close cooperation with WMO employees at the municipality of Groningen. Our experience confirmed that the translation of the requirements as expressed by non-technical employees to the representation required by our framework is rather intuitive, and is relatively easily understood when shown to non-experts for proof-checking.

In the tests presented in the next subsection, service invocations are simulated, and the methods provided by the service instances have a predefined behaviour, simulated according to the different situations we want to test. The performance of the framework has been tested with respect to atomic action repositories of increasing size, since domains that comprise a large set of actions, may raise concerns of inefficiency. All tests presented thereafter were performed on a computer with an Intel® Core™2 Duo processor @2,83GHz, with 3GB of RAM, running Java 1.6.0 24.

7.2.1 Tests on case study

In order to test the framework we have developed on a real case-study, the WMO process shown in Figure 3.6 was modelled, along with the DSs shown in Figure 5.6. The BP specification representing the case-study is as shown in 8.8, while the Planning Domain used by the planner is the output of Algorithm 1, given this BP specification and the set of atomic actions descriptions.

Table 7.1 provides an overview of the times required to generate the initial plans for all IPs shown in Figure 5.7, corresponding to DS1 of Figure 5.6, in case of a change in the applicant's address. In all cases, the time for generating the respective initial IP is below 1 second and, therefore, neglectable. However all IPs in this example, except for case (e), comprise one or more deferred choices, which implies that re-planning may be needed. As a result, after the execution of a knowledge-providing action, a violation check verifies whether the actual output differs from the expected value. If that is the case, the planner is invoked again with the same goal, but starting from the updated state corresponding to the newly sensed value(s).

IP	Plan length	Time for planning (in sec)
a	5	0.51
b	6	0.59
c	6	0.60
d	7	0.62
e	2	0.39

Table 7.1: Performance results for generating the IPs of Figure 5.7

Tables 7.2a and 7.2b present the times for computing each updated plan in the case of some possible environmental behaviour for the IPs depicted in Figures 5.7b and 5.7c, which have 2 and 3 deferred choices respectively. Re-planning is performed until the goal as specified in Section 6.2 is satisfied, or no solution can be found. The reported times are the average over 4 separate test runs.

The IP in Figure 5.7b corresponds to the situation where a change in address occurs when a wheelchair is already ordered but not yet delivered. The initial plan in Table 7.2a is generated assuming optimistic outcomes for the variables that are unknown at runtime. Consequently, it is assumed that no extra medical advice is required (*hvOut_medAdvReq=FALSE*) and that the decision is positive (*dcOut_decision= 'Approved'*). During execution of the initial plan, the PE may find out that a medical advice is required, in which case it updates the plan accordingly by including an extra action. If the outcome of the decision is negative, a constraint violation is encountered by the PE. The new situation (with *dcOut_decision= 'Not Approved'*) is sent to the planner for re-planning. In that case, however, no plan can be found that fulfills the goal, and the PE is informed accordingly.

The IP in Figure 5.7c covers the case where the address changes at the stage where a home modification is requested, but the request is not yet confirmed. Table 7.2b presents the times for the initial plan (assuming no medical advice, a positive decision, and the selected tender to be approved), and the potential updates as a result of re-planning. The actual service invocations may lead to the following discrepancies: the medical advice is actually required, and the plan is updated; the decision is negative, in which case no plan can be found that reaches the goal; the selected tender is not approved and a new plan is computed, asking the user to make a new selection (see also Section 6.3.4 for a possible execution behaviour showing the exact service invocations that take place).

State when planner is called	Plan length	Time for violation check and planning (in sec)
Initial state	6	0.62 (optimistic plan)
“Medical Advice required”	5	0.29 (violation, new plan)
“Rejected”	- (no plan)	0.02 (violation, goal cannot be satisfied)

(a)

State when planner is called	Plan length	Time for violation check and planning (in sec)
Initial state	6	0.61 (optimistic plan)
“Medical Advice required”	6	0.32 (violation, new plan)
“TenderNotOK”	4	0.19 (violation, new plan)
“Rejected”	- (no plan)	0.02 (violation, goal cannot be satisfied)

(b)

Table 7.2: Re-planning times for the IP of Figure 5.7b (a) and the IP of Figure 5.7c (b)

7.2.2 Scalability in a simulated domain

In the case of the WMO process, the planning domain comprises 16 actions (i.e. the BP-pertinent methods including both the actions that are part of the BP and the compensation actions), while the largest IP consists of 7 actions (note that if one adds up all actions that are executed as part of the re-planning process, the total number of actions that are executed as part of an IP may be significantly larger). For most BPs, the length of the IPs for recovering from the most usual situations are relatively short. However, there are occasions where the length of the required IPs might be significantly larger than the examples presented for the WMO case. For example, since the planner cannot produce plans with structured loops, many repetitions of a set of actions may be required to represent the desired pattern.

In order to evaluate the scalability of our framework with respect to the size of the required IPs (i.e. the number of activities they comprise), a number of tests have been performed with different goals, whose fulfillment requires IPs with an increasing size from 5 to 30 activities. For the sake of these tests, a virtual set of 100 atomic actions has been created, comprising the search space of the planner. The actions in the domain are interconnected through trivial sequence relations, so that all actions preconditions and effects are conjunctions of the same arity. The results of these tests are summarized in Table 7.3. They give an impression of how composition time is affected by the size of the required IP, for a given a business domain that consists only of sequence structures. The tests show that for a trivial domain, less than 6 sec are required to generate an IP comprising as many as 30 activities.

	5 act	**10 act**	**15 act**	**20 act**	**25 act**	**30 act**
Planning time (in sec)	4.89	5.12	5.21	5.33	5.51	5.69

Table 7.3: Performance results: Time for generating IPs of increasing size (domain size=100)

The time required to generate an IP is not only affected by the size of the domain, but also depends highly on the structure of both the planning domain, i.e. the interdependencies between the actions, and the goal. Disjunctive propositions resulting either from action preconditions or the goal (e.g. in cases where the under-condition goal construct is used), are known to add an extra burden to the constraint solver. Therefore, the most costly structures for the planner's performance are nested XORs with many branches (see Algorithm 3) and to a less extent the repeat structures leading to a disjunctive effect (see Algorithm 1). More information about the performance of the planner on different scenarios can be found in (Kaldeli et al., 2011). The experimental evaluation presented herein confirms that the time for generating an IP in realistic situations is a matter of a few seconds, which is an acceptable performance considering the average throughput time of long-running BPs (varying between 1 and 6 weeks for the WMO case).

CHAPTER 8

General discussion and conclusion

8.1 Introduction

The main purpose of this thesis is to investigate how business process interference can be identified and prevented in enterprise information systems. In order to achieve this objective, this research is divided into three parts, as described in detail in Section 1.3. Part I concerned the identification of process interference in business processes. The identification of specific interference cases and the analysis of the details of these cases assisted us in Part II, where a number of IT artefacts were developed, in order to prevent process interference and ensure correct process results. Finally, in Part III, the developed artifacts and corresponding IT architecture were implemented and tested on a case study in eGovernment. Consequently, we were able to evaluate the developed artifacts on performance and capability to resolve process interference.

In this final chapter, a reflection and discussion will be provided on the research in this thesis. The structure of this chapter is as follows. First, we will provide a discussion on the research process itself. Next, we will subsequently provide a detailed discussion on each of the three parts.

8.2 Reflection on the research process

Design science creates and evaluates IT artifacts to solve organizational problems (Hevner et al., 2004). The research presented in this thesis is triggered by a business problem, where the problem statement is a general formulation of the problem as perceived by organizations and their stakeholders. According to the model of Peffers et al. (2007), this research can be categorized as problem initiated design science.

In design science, seven distinct guidelines are specified for conducting and evaluating good research (Hevner et al., 2004). As design science is inherently a problem solving process, an innovative (and, therefore, novel), purposeful artifact must be created for a certain problem domain, to resolve a formerly unresolved problem. The established utility must be thoroughly evaluated. The presented research must be rigorously defined, formally represented, coherent, and internally consistent. Design science is an iterative process, using a heuristic search strategy to produce a feasible and good design that can be implemented in the business environment (Hevner et al., 2004). Finally, the results of the research (i.e. the designed artifacts) must be communicated effectively to a technical audience as well as a managerial audience.

The required artifacts to solve the problem can be constructs, methods, models, instantiations or better theories (March and Smith, 1995). In this research, the developed artifacts will be in the form of newly developed modelling constructs. The guidelines are summarized in Table 8.1 and are used for the evaluation in the following subsections accordingly.

Guideline	Description
Guideline 1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Table 8.1: Design-Science Research Guidelines (Source: (Hevner et al., 2004))

8.2.1 Design as an artifact

Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation. In this thesis, both a method and a modelling construct have been developed. An interference identification method and identification tool have been developed in Chapter 4. In addition, dependency scopes and intervention processes have been developed, along with the algorithms for automated specification (Chapter 5 and 6). Therefore, we consider the first guideline to be fulfilled.

8.2.2 Problem relevance

The research presented in this thesis is triggered by a business problem, where the problem statement is a general formulation of the problem as described by the anecdotal evidence of organizations and their stakeholders (Chapter 1). In addition, process interference as such has been acknowledged in literature, but has not yet been resolved satisfactorily (Chapter 2).

This is supported by the results of the analysis of the EC case and TC case, which shows that process interference is indeed widely spread in industry. The problem itself is important, as the effects on organizations and society can be considerable (Chapter 4).

8.2.3 Design evaluation

According to Guideline 3, the utility, quality, and efficacy of a design artifact must be evaluated. The evaluation of the designed artifact should include the integration of the developed modelling constructs in the business environment (Hevner et al., 2004). This environment includes the technical infrastructure, which constitutes the implementation of the developed modelling constructs. The business environments presented by the EC case and TC case established the requirements for the evaluation, whereas the WMO case was used as an expository instantiation (Gregor and Jones, 2007) in Chapter 7.

The developed modelling constructs are evaluated in terms of usability, accuracy, performance and reliability. The modelling constructs have been specifically designed to fit with current modelling standards. However, the first implementation revealed that manually defined intervention processes only provide a linear improvement as the complexity of the process increases (along with the number of dependency scopes). In order to improve the usability of the artifacts, automatic generation of intervention processes and critical sections has been facilitated.

The accuracy in runtime recovery from process interference has been evaluated by a simulation of the process with disruptions. The evaluation of the modelling constructs shows that the modelling constructs provide an accurate runtime recovery from process interference and perform well. In addition, the developed modelling constructs proved to be reliable in resolving various interference cases.

This way the requirements of Guideline 3 have been fulfilled.

8.2.4 Research contributions

According to Guideline 4, design-science research must provide clear and verifiable contributions in the areas of the artifact and design methodology. This thesis provides a problem identification technique (Chapter 4) as well as a problem solution technique (Chapter 5 and 6).

Prior to this research, process interference was acknowledged in academic literature, but a suitable solution for *all* cases was not provided. Using the artifacts developed in this thesis, process interference can be automatically resolved and, therefore, erroneous process outcomes can be prevented.

8.2.5 Research rigor

Guideline 5 stresses the application of rigorous methods in both the construction and evaluation of the design artifact. First, the existing knowledge base regarding the problem was investigated in Chapter 2. Existing academic literature is investigated to identify the extent to which process interference is solved with the current business process modelling techniques and to identify which constructs have already been developed that migth contribute to develop an integrated solution.

In Chapter 4, a method was developed and utilized for identifying the extent of process interference in two case studies. The resulting insights were used for the design of the artifacts (the modelling constructs) in Chapter 5 and Chapter 6. During the design of the artifacts, continuous improvements were made to the design while implementing and testing the solution on the WMO case.

The designed artifacts were finally evaluated in Chapter 7. The research process itself is evaluated in this chapter.

8.2.6 Design as a search process

According to Guideline 6, the search for an effective artifact requires the use of available instruments to reach a desired solution while satisfying all constraints in the problem environment. Design science is an iterative process, using a heuristic search strategy to produce a feasible and good design that can be implemented in the business environment (Hevner et al., 2004). This research fulfills Guideline 6 as follows.

First, two cases were analyzed to identify the extent of the problem, which can serve as a basis for the solution to be developed. The method developed and used for this analysis can be considered a design artifact as well. The second part of the research presented in this thesis comprises the design of the solution to the problem. By utilizing an iterative design process, the quality of process repair was improved in a second iteration.

The heuristic design solution is developed with a close proximity of an ”optimal” solution in mind. That is, for each disruption, the framework should (i) respond to that disruption, (ii) find a suitable intervention process and (iii) pose a minimal additional effort to the process designer. As such, the first priority was to establish that the developed solutions did work. Subsequently, the required effort for the process designer (i.e. the intended user of the developed artifact) was reduced by an iterative process of automation design for subsequent components of the designed artifact. This resulted in the automated generation of the planning domain (Section 6.3.2) and the automated generation of the critical sections (Section 6.4). The iterative process was facilitated by the expository instantiation of the WMO case. In Table 8.2, an overview is provided of the research instruments used in each phase of this research.

Activity	**Data source**	**Research instruments**
Initiation phase (Chapter 2)	Practitioners, literature	Interviews and literature review
Case study phase (Chapter 3)	Process experts, documentation, literature	Interviews, analysis of process documentation and literature
Method development phase (Chapter 4)	Transcribed interviews, structured documentation, literature, practical experience	Analysis of documentation and literature
Solution development phase (Chapter 5 and 6)	Formal definition of the problem from Phase I, practical experience	Interpretation of the problem as obtained from the case study and expository instantiation
Solution evaluation phase (Chapter 7)	Process output, execution time	Expository instantiation

Table 8.2: Overview of used research instruments

8.2.7 Communication of research

Although the presentation of this research is aimed at an audience familiar with business processes, workflows and data representations, the thesis also contains important, useful information for a managerial audience.

The output of this research is communicated through conference papers, journal papers, conference presentations and this thesis. This way, the problem, the problem identification process and the developed solution are provided to both technical and managerial audiences.

Consequently, Guideline 7 has been fulfilled.

8.3 Discussion on Part I: Process interference identification

In this part, a methodology is presented that allows to identify and analyze the potential inconsistency issues resulting from concurrently executed processes. Moreover, this methodology is applied to two distinct cases, showing the severity of the problem for these organizations.

8.3.1 Reflection on results

The analysis showed that concurrently executed processes indeed may interfere in practice. Furthermore, the validation with process experts revealed that the unknown problems as indicated by the analysis tool are common practice in reality as well. The amount and severity of the overlap identified confirms the presumed frequency of occurrence of the problems as well as the corresponding relevance for organizations.

The methodology showed its ability to efficiently provide a representative and valuable insight in the interference between concurrent processes and the potential disruptions. In addition, the methodology is applicable using semi-structured process documentation and does, therefore, not require the availability of a formal representation of the business process. Consequently, the provided results are legible by users without in-depth knowledge of implementation specifics.

8.3.2 Methodological considerations

From a methodological point of view, this part uses a triangulation strategy (Benbasat et al., 1987). The tool presented in Chapter 4 can be interpreted as a methodological instrument that has been applied in two case studies (i.e. EC and TC) to successfully identify process interference. This application of the tool verifies the technical implementation of the proposed methodology. Experts from both companies have been consulted to ascertain the practical relevance of the data-flow errors identified by the tool. This has further confirmed the added business value of the tool. Moreover, from a design science perspective, the established criteria for artifact-driven research (Gregor and Jones, 2007) have been satisfied.

Within this methodology-based context, one of the most important findings is that two of the seven identified cases of severe data-flow errors where known to the process experts. Highlighting these findings in front of the experts lent immediately more credibility to this work, and expanded their effort to identify symptoms and causes of the other tool-identified errors. Furthermore, revealing the basic nature of the error eased the finding of solutions for these errors.

8.3.3 Process interference vs. software

The application of the methodology reveals potential problems in concurrent process execution with shared data and identifies the potential interfering processes. Moreover, it provides insight in the severity of potential interference between concurrently executed processes and the variables that are involved, which provides the opportunity to resolve or prevent these situations in the Enterprise Information System.

The analysis has been performed independent from any implementation and does, therefore, not reveal whether the problems can be prevented by other measures, such as coordination or a better software implementation. Correspondingly, the cases of potential interference found in the analysis of EC and TC is not a result of a poor software implementation.

A vast majority of the identified erroneous cases concerned the problems responsible for many unresolved customer complaints that proved hard to diagnose. These complaints could not be resolved through existing techniques of process analysis and verification. In this respect, this part has gone beyond past research, by analyzing the process flow along with the information required in each of the distinct activities. As a result, the problems responsible for the unresolved customer complaints could be identified and located. The results of the application of the methodology to the case clearly show the importance and relevance of these business problems, as severe overlap in concurrent processes is widely spread.

In addition to the theoretical deduction of the potential consistency issues, this part contributes indirectly to the area of business intelligence as well. If data-flow errors remain undetected, business strategies formulated from mining transactional data would be ineffective. For example, if organizations are planning to tailor business strategies according to the geographical distribution of customers, then inaccurate addresses would translate to wrongful interpretations of consumer preferences. Therefore, the methodology does not only improve operational efficacies (i.e., better customer service), but it also augments strategic decision making (i.e., data mining in formulating business strategies).

8.4 Discussion on Part II: Concepts definition and automation

In this part, an approach is presented for automated runtime process repair in case of interference, which ensures the recovery of a BP from erroneous states without the necessity of predefining all potential interference situations, and the respective ways to overcome them.

8.4.1 Reflection on developed artifacts

For that purpose, dependency scopes are defined to represent the dependencies between processes and data sources. In addition, intervention processes are developed to repair erroneous path situations using dynamic reconfiguration during execution of the process. We have shown that both dependency scopes and intervention processes can easily be integrated within an existing BPMS platform.

Dependency scopes

The correct identification of the sections of a business process, whose correct execution depends on some volatile variable, is very important. These sections should be guarded upon, so that whenever a modification event is received during their execution, an appropriate intervention process is executed, in order to restore the process to a consistent state. However, the task of manual specification of these critical sections can become cumbersome and prone to errors, especially for processes with a complex structure, using many shared resources. To facilitate this task, an algorithm was developed, which automatically computes the appropriate critical sections, given a BP specification and some semantics regarding the input-output and the internal state variables of the service operations used by the process.

Intervention processes

For complex processes, it is unfeasible to specify the appropriate intervention processes manually, as this can be particularly time-consuming and error-prone, while it is difficult to ensure that all important intervention cases are taken into account. Therefore, an approach for automating the generation of intervention processes at runtime was proposed, by using domain-independent AI planning techniques. This way, intervention processes are composed on the fly, taking into account the characteristics of the business process in execution, the available compensation activities, and the properties that have to be fulfilled to recover from the erroneous situation. As such, we show how AI planning can be used to ensure that the consistency of the process execution results in an automatic way.

In this thesis, we have mainly concentrated on process interference situations between different processes, as this is the most typical in practice. However, the problem of process interference is not necessarily a single instance problem. For example, an order may consist of multiple order lines, deliveries may group different orders etc. Although not explicitly presented in this thesis, our approach may also be applied to check if the process interferes with itself. The framework developed in this thesis considers data overlap, which causes interference. If data overlap occurs with multiple process instances resulting in interference, the AI planner will provide a solution, regardless whether the processes are essentially the same.

However, a situation might occur that does require intervention, but the AI planner is not able to generate an intervention process that fulfills the goal. In these cases, two possible solutions can be suggested. First, the process can be paused and require a human decision on how to proceed. Second, a rollback can be executed. This is, however, the least desirable solution, especially in processes with a long lead time.

8.5 Discussion on Part III: Implementation and evaluation

8.5.1 Interference resilience

To evaluate the feasibility of the approach, an architecture has been designed and a prototype has been implemented. The WMO process of the eGovernment case study was implemented with the prototype and the execution of the process was simulated. A number of deliberate disruptions were inserted during execution, in order to test the resilience of the developed IT artifacts on external data changes and, therefore, to test the solution for process interference.

The results indicate that coupling DSs with declarative goals and generating IPs at runtime by means of AI planning is a usable and realistic method for resolving erroneous path situations caused by process interference. The proposed method is both sound and complete. That is, the generated IPs always satisfy the properties specified in the goal, and if there exists a combination of activities that achieves the goal, then this sequence is found.

The IP generated is finite in all cases. Although generated IPs may include finite loops through an enumerated repetition of certain activities, they cannot include indefinite loops, since a plan provided by the AI planner is a finite, partially ordered set of actions. However, if the output of deferred choices continuously satisfies the loop condition, the AI planner may be called indefinitely. In order to avoid such situations, an upper limit is put to the number of times the re-planning process can be invoked.

8.5.2 Performance

Apart from the quality of the generated IPs, the performance of the AI planner has been evaluated as well. First, the planning time for a large number of IPs was measured for the WMO process. Next, the scalability of the framework was evaluated with respect to the size of the required IPs (i.e. the number of activities they comprise), as the time required to generate an IP is not exclusively dependent on the structure of the planning domain (i.e. the interdependencies between the actions, and the goal). For that purpose, a number of tests have been performed with different goals, whose fulfillment requires IPs with an increasing size from 5 to 30 activities.

The performance evaluation shows that the time required for generating an IP in realistic situations is a matter of a few seconds, which is an acceptable performance considering the average throughput time of long-running BPs (varying between 1 and 6 weeks for the WMO case).

8.6 Reflection, limitations and further research

8.6.1 Reflection on available expertise

As mentioned in Chapter 1, this thesis is a joint work with the department of Distributed Systems of the University of Groningen. The work presented in this thesis required experts in different fields with respect to the AI techniques and formalisms. Part I relies on the correctness of the formal process interference definition. Consequently, we acquired the required expertise from a model checking expert, Doina Bucur, for verifying the correctness of the formalisms presented in Chapter 4. Additionally, we have collaborated with Eirini Kaldeli and Pavel Bulanov for obtaining the knowledge regarding the AI planning techiques and the service-oriented implementation of the architecture.

8.6.2 Reflection on the solution

Initially, this research focussed on process and data integration, however, case studies indicated that the problem could not be resolved in this way. An integration of data with the process specifications requires data to be a fundamental part of workflow verification. This implies that all data changes initiated by processes outside the scope of the process model are still not incorporated as part of the process design and its exception handling. As a result, both the identification and runtime solution of process interference cannot be provided using such an integration.

It appears that the process interference problem cannot be resolved without the concept of a dependency scope. However, a number of alternatives can be identified with respect to the algorithm for transforming the business process specification into a planning domain as well as the planner itself.

The domain generator algorithm, as presented in Section 6.3.2, takes the preconditions regarding deferred choices and loops into account, in order to ensure that the generated intervention processes are still compatible with the business rules in the original process. However, the explicit process structures were not incorporated as this would pose a too strong restriction on the generated intervention process. That is, if certain preconditions occur in a loop, the loop itself should not necessarily be preserved by means of additional preconditions. Consequently, process structures are only implicitly preserved by preconditions of the first activities in those structures.

Within the same architecture several techniques for generating intervention processes can be identified, some of which have been reviewed in Section 2.4. However, the CSP-planner used in this thesis is domain independent and supports extended goals, including temporal goals and maintainability. Nevertheless, the generation of intervention processes is not limited by the AI techniques used in this thesis.

8.6.3 Limitations

The application of the methodology reveals potential problems in concurrent process execution with shared data. Obviously, it does not reveal whether the problems can be prevented by other measures, such as coordination or a better software implementation. Furthermore, the methodology does not show the exact erroneous output or the implications for reality. Rather, application of the methodology only identifies the potential interfering processes. Moreover, it provides insight in the severity of potential interference between concurrently executed processes and the variables that are involved.

Concerning the designed artifacts, an intervention process is only generated and executed in case of change events that are covered by dependency scopes. Although dependency scopes are automatically identified, still a dependency might be overlooked due to changes that are not timely reported by users or customers. Consequently, the new business reality may result in errors that are not captured by the framework.

Furthermore, a situation might occur that does require intervention, but the AI planner is not able to generate an intervention process that fulfills the goal. In these cases, two possible solutions can be suggested. First, the process can be paused and require a human decision on how to proceed. Second, the entire process can be reverted, which includes cancelling all orders etc. This is, however, the least desirable solution, especially in processes with a long lead time.

8.6.4 Directions for future research

Foreseen future research for the identification methodology can be described as follows. The development of a context-independent categorization of data can contribute to the generalizability of the method. Essential data can be defined and represented in terms of data hierarchies, indicating the importance of data for the smooth running of one or more business processes within and/or across corporate hierarchies. In defining essential data according to data hierarchies, it might be possible to incorporate additional tracing capabilities into the tool that enable process experts to trace the impact caused by specific data-flow errors.

Foreseen future research for the designed artifacts can be described as follows. Although the focus of this thesis is to deal with inconsistencies that result from process interference, the overall approach based on domain-independent AI planning for BP reconfiguration is more general. For example, the system can be extended so that it can be used for process adaptation in case of changes in the business requirements/rules. The dynamic nature of the CSP-based planning framework allows the incorporation of changes in the BP-specific constraints at runtime: constraints which become obsolete can be removed on-the-fly from the constraint network, and the same holds for the addition of new constraints. It should be noted that the precondition and effects language used in the service descriptions is in line with existing semantic markups for Web Services such as OWL-S. Finding a suitable and yet powerful interface for designing goals and service descriptions and integration with existing standards is open for future investigation.

8.7 Answer to the research questions

RQ1: How can business process interference be identified?
Process interference is defined in this thesis as the situation where data modications by one process affect one or more other concurrently executing processes, which potentially causes an undesired process outcome for one or more of these processes.

Process interference can be identified by a combinatorial analysis of concurrent business processes by comparing the desired outcome and the outcome as provided by a certain execution combination.

RQ2: How can the severity of existing business process interference be assessed?
The severity of the erroneous outcomes resulting from process interference is defined by both the number of possible erroneous execution combinations and the nature of the erroneous outcome itself. A large number of possible erroneous execution combinations implies that concurrent execution of the processes under inves-

tigation have a large probability to provided an erroneous outcome. Furthermore, it is potentially more harmful if the value of a certain process variable is not only different from the desired situation, but also originates from a different stakeholder than in the desired situation.

Consequently, the severity of existing business process interference can be assessed using the tool, by analyzing both the number of erroneous situations and the actual values.

RQ3: How can business process interference be prevented in enterprise information systems and which artefacts are required to ensure process and data consistency?
Erroneous process outcomes are the result of erroneous path situations, which are a consequence of process interference. By identifying and explicitly representing the dependencies between processes and data sources, the potential occurrence of erroneous path situations can be intercepted. Consequently, if the external data change indeed would cause a potentially erroneous outcome, the currently executed process can be dynamically reconfigured to resolve the potentially troublesome situation.

As such, process interference (more specifically, the potentially undesired process outcomes) can be prevented by defining explicit dependencies between processes and data sources along with dynamic runtime process repair by means of automatic reconfiguration.

The main artefacts developed in this thesis are *Dependency Scopes (DS)* and *Intervention Processes (IP)*. Dependency scopes are defined to represent the dependencies between processes and data sources. Intervention processes are developed to repair erroneous path situations using dynamic reconfiguration during execution of the process.

RQ4: What techniques are required for automated recovery from process interference?
Manual specification of dependency scopes and intervention processes poses a significant workload on the process designer. In addition, manual specification is prone to errors, due to the complexity of the processes and their interactions with the environment.

For that reason, techniques are required to automate the specifcation of both dependency scopes and intervention processes. As such, an algorithm for *automatic identification of dependency scopes* has been developed, which is capable of generating the appropriate dependency scopes based on the process description and data specification. In addition a *Domain Generator* has been developed to enable such automatic DS composition and automated composition of the planning domain, which is required for identifying the available activities for the intervention process. Subsequently, a *CSP-based planner* will generate the intervention process required to resolve the erroneous situation. The aforementioned techniques together allow for automated revovery from process interference.

8.8 Implications for organizations

In the real world, the errors caused by process interference lead to customer complaints, legal cases, and many untraceable societal costs (Van Beest et al., 2010b). Although the process provides erroneous results in such cases, no immediate software errors occur. Consequently, the incorrect impression exists that the process runs well. As a result, the origin of these unresolved customer complaints have proven to be hard to diagnose and their root cause, process interference, is overlooked in process management software architectures.

The developed methodology showed its ability to efficiently provide a representative and valuable insight in the interference between concurrent processes and the potential disruptions emerging. The application of the developed methodology to two case studies clearly indicated the severity and importance of the problem. For *all* analyzed process pairs, a significant number of combinations resulted in serious disruptions in the process outcomes. The developed methodology clearly contributes to the identification and diagnosis of the problems that result from process interference. As such, organizations are provided with a tool that allows for a clear analysis of those processes that require additional measures to prevent the customer complaints resulting from process interference.

Using the developed artifacts, process interference can be resolved in an automated way and, therefore, the described erroneous process outcomes can be prevented. The application of the approach in business domains where data can be changed by external factors, can be highly beneficial for organizations, particularly considering the pervasiveness of the problem. Potential inconsistencies are resolved before actual erroneous outcomes are provided to the customer. The aforementioned customer complaints and legal cases can, for that reason, be prevented. Furthermore, potential inconsistencies are resolved in a way that enables a higher degree of flexibility by reducing hard-coded dependency solutions and workflow repair mechanisms. Due to the full automated support of the developed artifacts, the provided solution does not require additional effort from the process designer.

In addition, the application of the framework will be beneficial to those organizations that tend to change their business processes rather frequently. As a result of the automated dependency checks and repair processes, it can be expected that flexibility increases, as the amount of manually specified exception handling processes can be reduced. Consequently, deployed EISs will pose a smaller constraint on organizational agility.

In summary, an important and widespread business problem is addressed and resolved in this thesis, while preserving business process flexibility and minimizing developer effort.

Bibliography

M. Aiello and A. Lazovik. Monitoring assertion-based business processes. *International Journal of Cooperative Information Systems*, 15:359–389, 2006.

A.A. Alwan, Ibrahim H., and N.I. Udzir. A framework for checking and ranking integrity constraints in a distributed database. *Journal of Next Generation Information Technology*, 2:37–48, 2011.

A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. Liu, S. Thatte, P. Yendluri, and A. Yiu. Web services business process execution language version 2.0, 2007. WS-BPEL TC OASIS, April 2007.

T. Au, U. Kuter, and D. Nau. Web Service Composition with Volatile Information. In *Proceedings of the 4th International Semantic Web Conference (ISWC)*, pages 52–66, 2005.

H. Balsters and G.B. Huitema. Semantics of interoperable and outsourced information systems. In *Enterprise Interoperability*, Lecture Notes in Computer Science, pages 13–22. Springer London, 2007. ISBN 978-1-84628-714-5.

C. Beckstein and J. Klausner. A meta level architecture for workflow management. *Journal of Integrated Design and Process Science*, 3:15–26, 1999.

I. Benbasat, D.K. Goldstein, and M. Mead. The case research strategy in studies of information systems. *MIS Quarterly*, 11:369–386, 1987.

P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

H.H. Bi and J.L. Zhao. Mending the lag between commercial needs and research prototypes: A logic-based workflow verification approach. In *8th INFORMS Computing Society Conference*, pages 191–212, 2003.

G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide (2nd Edition)*. Addison-Wesley Professional, 2005. ISBN 0321267974.

T.D. Bouma. Process analysis and requirement specification of software as service for WMO provision applications at dutch municipalities, 2010.

D. Bucur and M. Kwiatkowska. On software verification for sensor nodes. *Journal of Software and Systems*, 84:1693–1707, 2011.

E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK, 1999.

P. Dadam and M. Reichert. The ADEPT project: a decade of research and development for robust and flexible process support. *Computer Science - R & D*, 23: 81–97, 2009.

T.H. Davenport and J.E. Short. The new industrial engineering: information technology and business process redesign. *Sloan Management Review*, 31:11–27, 1990.

M. De Leoni, M. Mecella, and G. De Giacomo. Highly dynamic adaptation in process management systems through execution monitoring. *BPM 2007*, pages 182–197, 2007.

M De Leoni, G. De Giacomo, Y. Lespèrance, and M. Mecella. On-line adaptation of sequential mobile processes running concurrently. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1345–1352. ACM, 2009.

T. Dewett and G.R. Jones. The role of information technology in the organization: a review, model, and assessment. *Journal of Management*, 27:313–347, 2001.

R.M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in bpmn. *Information and Software Technology*, 50(12):1281–1294, 2008.

E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8:569, 1965.

E.A. Emerson and J.Y. Halpern. ”sometimes” and ”not never” revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33:151–178, 1986.

H.M. Ferreira and D.R. Ferreira. An integrated life cycle for workflow management based on learning and planning. *International Journal of Cooperative Information Systems*, 15:485–505, 2006.

M. Fowler and K. Scott. *UML distilled (2nd edition): a brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0-201-65783-X.

M. Gajewski, H. Meyer, M. Momotko, H. Schuschel, and M. Weske. Dynamic failure recovery of generated workflows. *Database and Expert Systems Applications, International Workshop on*, pages 982–986, 2005.

H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of 1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259. ACM, 1987.

M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

K. Göser, M. Jurisch, H. Acker, U. Kreher, M. Lauer, S. Rinderle, M. Reichert, and P. Dadam. Next-generation process management with adept2, 2007.

S.D. Gregor and D. Jones. The anatomy of a design theory. *Journal of the Association for Information Systems*, 8(5):312–335, 2007.

T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–318, 1983.

M. Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173:503–535, 2009.

M. Henneberger, B. Heinrich, F. Lautenbacher, and B. Bauer. Semantic-based planning of process models. In *Multikonferenz Wirtschaftsinformatik (MKWI)*. GITO-Verlag, Berlin, 2008.

A.R. Hevner. A three cycle view of design science research. *Scandinavian Journal of Information Systems*, 19(2):87–92, 2007.

A.R. Hevner, S.T. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

J. Hoffmann, I. Weber, and F. Kraft. SAP Speaks PDDL: Exploiting a software-engineering model for planning in business process management. *Journal of Artificial Intelligence Research*, 44:587–632, 2012.

P. Jarvis, J. Moore, J. Stader, A. Macintosh, A. Casson-du Mont, and P. Chung. Exploiting ai technologies to realise adaptive workflow systems. In *In Proceedings of the AAAI Workshop on Agent-Based Systems in the Business Context, 1999*. AAAI Technical Report WS-99-02., 1999.

K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modeling and Validation of Concurrent Systems*. Springer-Verlag Berlin, 2009. ISBN 9783642002830.

P. Johannesson and E. Perjons. Design principles for process modelling in enterprise application integration. *Information Systems*, 26(3):165–184, 2001.

M.B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition*. Packt Publishing, 2006. ISBN 1904811817.

E. Kaldeli, A. Lazovik, and M. Aiello. Extended goals for composing services. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*. AAAI Press, 2009.

E. Kaldeli, A. Lazovik, and M. Aiello. Continual planning with sensing for Web Service composition. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (to appear)*. AAAI Press, 2011.

O. Kopp, D. Martin, D. Wutke, and F. Leymann. On the choice between graph-based and block-structured business process modeling languages. In *Modellierung be-*

trieblicher Informationssysteme (MobIS 2008), volume 141 of *Lecture Notes in Informatics (LNI)*, pages 59–72. Gesellschaft für Informatik e.V. (GI), 2008.

H.F. Korth and G. Speegle. Formal models of correctness without serializability. In *Proceedings of 1988 ACM SIGMOD International Conference on Management of Data*, pages 379–386. ACM, 1988.

L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5:1–11, 1987.

H.J. Levesque, R. Reiter, Y. Lesprance, F. Lin, and R.B. Scherl. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(13):59–83, 1997.

H. Li and Y. Yang. Dynamic checking of temporal constraints for concurrent workflows. *Electronic Commerce Research and Applications*, 4(2):124–142, 2005.

Y. Liu, S. Müller, and K. Xu. A static compliance-checking framework for business process models. *IBM Systems Journal*, 46:335–361, 2007.

T. Madhusudan, J.L. Zhao, and B. Marshall. A case-based reasoning framework for workflow model management. *Data Knowledge Engineering*, 50:87–115, 2004.

S.T. March and G.F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15:251–266, 1995.

A. Marrella and M. Mecella. Continuous planning for solving business process adaptivity. In *12th International Working Conference on Business Process Modeling, Development and Support (BPMDS 2011), in conjunction with CAiSE 2011*, 2011.

J. Martin. *Managing the Data-base Environment*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983. ISBN 0-13-550582-8.

J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. volume 4, pages 463–502. Edinburgh University Press, 1969.

H.S. Meda, A.K. Sen, and A. Bagchi. On detecting data flow errors in workflows. *Journal of Data and Information Quality*, 2:1–31, July 2010.

Welfare Ministry of Health and Sport. Wet maatschappelijke ondersteuning (wmo), 2008. 29-11-2008; (http://www.minvws.nl/dossiers/wmo/).

D. Moitra and J. Ganesh. Web services and flexible business processes: towards the adaptive enterprise. *Information & Management*, 42:921–933, 2005.

G. Monakova, F. Leymann, S. Moser, and K. Schäfers. Verifying business rules using an smt solver for bpel processes. In *Business Process and Services Computing Conference: BPSC'09*, pages 81–94, 2009.

R. Müller, U. Greiner, and E. Rahm. Agentwork: a workflow system supporting rule-based workflow adaptation. *Data and Knowledge Engineering*, 51:223–256, 2004.

T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, 1989.

L. Măruşter and N.R.T.P. Van Beest. Redesigning business processes: a methodology based on simulation and process mining techniques. *Knowledge and Information Systems*, 21:267–297, 2009.

J.M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.

OMG. Unified modeling language: Infrastructure, version 2.0, 2005. Object Management Group (OMG), Document Number formal/05-07-05.

OMG. Business process model and notation beta 1 for version 2.0, 2009.

C. Ouyang, M. Dumas, A.H.M. Ter Hofstede, and W.M.P. Van Der Aalst. From BPMN process models to BPEL web services. In *International Conference on Web Services (ICWS)*, pages 285–292, 2006.

C. Ouyang, E. Verbeek, S. Breutel, M. Dumas, and A.H.M. Ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67:162–198, 2007.

K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007.

C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, Germany, 1962.

M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *19th Int. Joint Conference on Artificial Intelligence*, pages 1252–1259, 2005.

K. Ramamritham and P.K. Chrysanthis. In search of acceptability criteria: Database consistency requirements and transaction correctness properties. In *Distributed Object Management*, pages 212–230. Morgan Kaufmann, 1993.

S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems a survey. *Data and Knowledge Engineering*, 50:9–34, 2004.

M.D. Rodríguez-Moreno and P. Kearney. Integrating ai planning techniques with workflow management system. *Knowledge-Based Systems*, 15(5-6):285–291, 2002.

M.D. Rodríguez-Moreno, D. Borrajo, A. Cesta, and A. Oddi. Integrating planning and scheduling in workflow domains. *Expert Systems and Applications*, 33(2): 389–406, 2007.

N. Russell, A.H.M. Ter Hofstede, D. Edmond, and W.M.P. Van Der Aalst. Workflow data patterns. Technical Report FIT-TR-2004-01, Queensland University of Technology, 2004. QUT Technical report.

N. Russell, A.M.H. Ter Hofstede, D. Edmond, and W.M.P. Van Der Aalst. Workflow data patterns: Identification, representation and tool support. In *Conceptual Modeling – ER 2005*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-29389-7.

N. Sidorova, C. Stahl, and N. Trčka. Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. *Information Systems*, 36(7):1026–1043, 2011.

S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison-Wesley, Reading, MA, 1990.

S. Sohrabi and S.A. McIlraith. Preference-based web service composition: A middle ground between execution and search. In *Proceedings of 9th International Semantic Web Conference (ISWC)*, pages 713–729, 2010.

S.X. Sun, J.L. Zhao, J.F. Nunamaker, and O.R.L. Sheng. Formulating the data-flow perspective for business process management. *Information Systems Research*, 17:374–391, 2006.

N. Trčka, W.M.P. Van Der Aalst, and N. Sidorova. Data-flow anti-patterns: Discovering data-flow errors in workflows. In *Proceedings of the 21st International Conference on Advanced Information Systems Engineering - CAiSE '09*, volume 5565 of *Lecture Notes in Computer Science*, pages 425–439. Springer-Verlag, 2009.

S.D. Urban, L. Gao, R. Shrestha, and A. Courter. The dynamics of process modeling: New directions for the use of events and rules in service-oriented computing. In *The Evolution of Conceptual Modeling*, volume 6520 of *LNCS*, pages 205–224. Springer-Verlag, 2011.

N.R.T.P. Van Beest, P. Bulanov, J.C. Wortmann, and A. Lazovik. Resolving business process interference via dynamic reconfiguration. In *Proceedings of 8th International Conference on Service Oriented Computing (ICSOC)*, pages 47–60. Springer, 2010a.

N.R.T.P. Van Beest, N.B. Szirbik, and J.C. Wortmann. Assessing the interference in concurrent business processes. In *Proceedings of 12th International Conference on Enterprise Information Systems (ICEIS)*, pages 261–270, 2010b.

W.M.P. Van Der Aalst. Verification of workflow nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, ICATPN '97, pages 407–426. Springer-Verlag, 1997.

W.M.P. Van Der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8:21–66, 1998.

W.M.P. Van Der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14:5–51, 2003a.

W.M.P. Van Der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14:5–51, 2003b. ISSN 0926-8782.

W.M.P. Van Der Aalst, B. Van Dongen, J. Herbst, L. Măruşter, G. Schimm, and A. Weijters. Workflow mining: A survey of issues and approaches. *Data and Knowledge Engineering*, 47:237–267, 2003c.

W.M.P. Van Der Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - R & D*, 23:99–113, 2009.

P.J. Van Strien. Towards a methodology of psychological practice, the regulative cycle. *Theory and Psychology*, 7:683–700, 1997.

H.M.W. Verbeek. Analyzing BPEL processes using petri nets. In *Florida International University*, pages 59–78, 2005.

H.M.W. Verbeek, T. Basten, and W.M.P. Van Der Aalst. Diagnozing workflow processes using woflan. *The Computer Journal*, 44:246–279, 2001.

I. Weber, J. Hoffmann, and J. Mendling. Beyond soundness: on the verification of semantic business process models. *Distr. and Parallel Databases*, 27:271–343, 2010.

M. Weske. Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, 2001.

M. Weske, G. Vossen, and F. Puhlmann. Workflow and service composition languages. In Peter Bernus, Kai Mertins, and Gnter Schmidt, editors, *Handbook on Architectures of Information Systems*, International Handbooks on Information Systems, pages 369–390. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-25472-0.

WfMC. The workflow management coalition specification, terminology and glossary, 1999. Document Number WFMC-TC-1011.

S.A. White. Business process modeling notation (bpmn) version 1.0., 2004. Business Process Management Initiative, BPMI.org.

Y. Xiao and S.D. Urban. Process dependencies and process interference rules for analyzing the impact of failure in a service composition environment. In *Proceedings of the 10th International Conference on Business Information Systems*, volume 4439 of *Lecture Notes in Computer Science*, pages 67–81. Springer Berlin / Heidelberg, 2007.

Y. Xiao and S.D. Urban. Using data dependencies to support the recovery of concurrent processes in a service composition environment. In *Proceedings of the 16th International Conference on Cooperative Information Systems*, pages 139–156, 2008.

Y. Xiao, S.D. Urban, and S. Dietrich. A process history capture system for analysis of data dependencies in concurrent process execution. In *Data Engineering Issues in E-Commerce and Services*, volume 4055 of *Lecture Notes in Computer Science*, pages 152–166. Springer Berlin / Heidelberg, 2006.

Appendices

A. Analysis results

In this Appendix, the low-level results of the analysis presented in Chapter 4 are presented. First, an explanation is provided of the low-level metadata as used by the analysis tool.

The first digit of each value always represents the stakeholder. The next two digits hold the count of WRITE assigments, which increment for every WRITE execution. Table 8.3 provides an overview of the values of d known at each stakeholder at the different states of the process shown in Figure 8.1. The initial values of d are set to 100, 200 and 300 for S1, S2 and S3 respectively. S1 executes A to read the value of d from S2. S1 and S2 now both hold the value 200. Next, S1 executes B to request an update of d. Therefore $[1]$ is added to the write sequence, to indicate that $process_1$ writes a new value to d. Note that in this case there is only one process, but the added value of this annotation in case of two or more processes is clear: the write sequence represents the order of write operations to a datafield. In addition, 200 is changed into 20$\underline{1}$, to indicate that this is the first WRITE activity. This number will increase at each WRITE activity. As a result, S1 has a different value of d than S2 and S3, as S2 sent the write request to S3. Hence, S1 did not receive this update and holds an outdated value.

Note that the first digit of the value of d never changes as a result of a WRITE operation, as this digit stores the stakeholder this value originates from (hence the

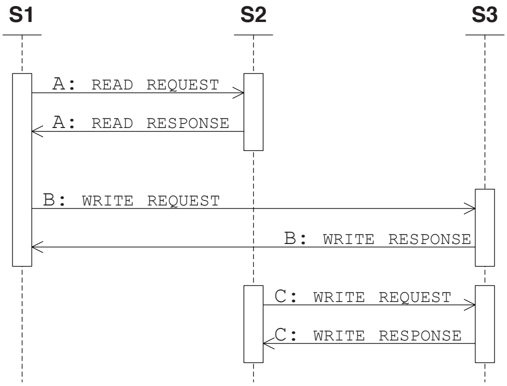


Figure 8.1: Sequence Diagram showing READ and WRITE services between stakeholders.

initial values of 100, 200 and 300). After execution of C, the first digit of the value at S1 equals 2 (2̲01). This implies that the value known to S1 is read from S2 prior to the WRITE operation. This way, the origin of a value can be traced at the final value.

	Initial value	**Value after A**		**Value after B**		**Value after C**	
Stakeholder	Value	Value	Write Seq	Value	Write Seq	Value	Write Seq
S1	100	200		201	[1]	201	[1]
S2	200	200		200		202	[1]
S3	300	300		201	[1]	202	[1]

Table 8.3: Value of d at different states in the process.

EC - First comparison: Change of Supplier – Move Out

In this comparison, the situation is analyzed where a person changes his energy provider and decides to move to a new address at about the same time. Three datafields are traced: *Supplier*, *New Supplier*, and *Address*. The initial values of the first comparison are provided in Table 8.4 below.

The desired outcome is obtained by executing *Change of Supplier* and *Move Out* sequentially, which is shown in Table 8.5. In Table 8.6, an example is provided of the erroneous output represented by Table 4.6 in Chapter 4.

Stakeholder	Supplier	New Supplier	Address
CCP	100	100	100
EMP	200	200	200
GridOperator	300	300	300
MRParty	400	400	400
NewPVShipper	500	500	500
NewSupplier	600	600	600
OldPVShipper	700	700	700
OldSupplier	800	800	800
TM2010	900	900	900

Table 8.4: Initial values 1st comparison EC case.

	Supplier		**New Supplier**		**Address**	
Stakeholder	Value	Write Seq	Value	Write Seq	Value	Write Seq
CCP	100		100		100	
EMP	218	[1][2]	615	[1][1][1][2]	819	[2][2]
GridOperator	218	[1][2]	300		819	[2][2]
MRParty	218	[1][2]	606	[1][1][1]	819	[2][2]
NewPVShipper	207	[1]	606	[1][1][1]	608	[1][1]
NewSupplier	211	[1][1][1][1]	606	[1][1][1]	608	[1][1]
OldPVShipper	700		615	[1][1][1][2]	700	
OldSupplier	822	[2][2][2]	615	[1][1][1][2]	813	[2]
TM2010	822	[2][2][2]	900		900	

Table 8.5: Desired output 1st comparison EC case.

	Supplier		**New Supplier**		**Address**	
Stakeholder	Value	Write Seq	Value	Write Seq	Value	Write Seq
CCP	100		100		100	
EMP	215	[1][2]	611	[1][1][2][1]	816	[2][1][2]
GridOperator	215	[1][2]	300		816	[2][1][2]
MRParty	215	[1][2]	611	[1][1][2][1]	816	[2][1][2]
NewPVShipper	215	[1][2]	611	[1][1][2][1]	816	[2][1][2]
NewSupplier	221	[1][2][1][1][1]	611	[1][1][2][1]	816	[2][1][2]
OldPVShipper	700		611	[1][1][2][1]	700	
OldSupplier	222	[1][2][1][1][1][2]	611	[1][1][2][1]	807	[2]
TM2010	222	[1][2][1][1][1][2]	900		900	

Table 8.6: Erroneous output 1st comparison EC case.

EC - Second comparison: Change of Supplier – Meter Change

In this comparison, the situation is analyzed where a person changes his energy provider and his meter is to be changed at about the same time. Three datafields are traced: *Supplier*, *Meter Reading*, and *Address*. The initial values of the second comparison are provided in Table 8.7.

Stakeholder	**Supplier**	**Meter Reading**	**Address**
EMP	200	200	200
GridOperator	300	300	300
MRParty	400	400	400
NewPVShipper	500	500	500
NewSupplier	600	600	600
OldPVShipper	700	700	700
OldSupplier	800	800	800
PVShipper	900	900	900
Supplier	1000	1000	1000
TM2010	1100	1100	1100

Table 8.7: Initial values 2nd comparison EC case.

The desired outcome is obtained by executing *Change of Supplier* and *Meter Change* sequentially. The output is shown in Table 8.8.

	Supplier		**Meter Reading**		**Address**	
Stakeholder	Value	Write Seq	Value	Write Seq	Value	Write Seq
EMP	206	[1]	311	[2]	607	[1][1][1][1]
GridOperator	300		300		607	[1][1][1][1]
MRParty	206	[1]	400		607	[1][1][1][1]
NewPVShipper	206	[1]	500		607	[1][1][1][1]
NewSupplier	206	[1]	1110	[1][1][1][1][1]	607	[1][1][1][1]
OldPVShipper	206	[1]	700		700	
OldSupplier	206	[1]	800		800	
PVShipper	206	[1]	900		607	[1][1][1][1]
Supplier	206	[1]	314	[2][2][2][2]	607	[1][1][1][1]
TM2010	1100		314	[2][2][2][2]	1100	

Table 8.8: Desired output 2nd comparison EC case.

In Table 8.9, an example is provided of the erroneous output represented by Table 4.7 in Chapter 4.

	Supplier		Meter Reading		Address	
Stakeholder	Value	Write Seq	Value	Write Seq	Value	Write Seq
EMP	207	[1]	301	[2]	608	[1][1][1][1]
GridOperator	300		300		608	[1][1][1][1]
MRParty	207	[1]	400		608	[1][1][1][1]
NewPVShipper	207	[1]	500		608	[1][1][1][1]
NewSupplier	207	[1]	312	[2][2][1]	608	[1][1][1][1]
OldPVShipper	207	[1]	700		700	
OldSupplier	207	[1]	800		800	
PVShipper	207	[1]	900		608	[1][1][1][1]
Supplier	200		314	[2][2][1][2][2]	200	
TM2010	1100		314	[2][2][1][2][2]	1100	

Table 8.9: Erroneous output 2nd comparison EC case.

EC - Third comparison: Change of Metering Responsible – Move Out

In this comparison, the situation is analyzed where the metering responsible is changed for a certain contract and the owner of that contract decides to move to a new address at about the same time. Three datafields are traced: *Current MR*, *New MR*, and *Address*. The initial values of the third comparison are provided in Table 8.10 below.

Stakeholder	Current MR	New MR	Address
EMP	100	100	100
GridOperator	200	200	200
MRParty	300	300	300
NewMRParty	400	400	400
OldMRParty	500	500	500
OldPVShipper	600	600	600
OldSupplier	700	700	700
PVShipper	800	800	800
Supplier	900	900	900

Table 8.10: Initial values 3rd comparison EC case.

The desired outcome is obtained by executing Change of Metering Responsible and Move Out sequentially. The output is shown in Table 8.11.

In Table 8.12, an example is provided of the erroneous output represented by Table 4.8 in Chapter 4.

	Current MR		New MR		Address	
Stakeholder	Value	Write Seq	Value	Write Seq	Value	Write Seq
EMP	107	[1][2]	108	[1][1][2]	709	[2][2]
GridOperator	200		200		709	[2][2]
MRParty	300		300		709	[2][2]
NewMRParty	102	[1]	104	[1][1]	405	[1][1]
OldMRParty	102	[1]	104	[1][1]	500	
OldPVShipper	107	[1][2]	108	[1][1][2]	600	
OldSupplier	107	[1][2]	108	[1][1][2]	706	[2]
PVShipper	800		800		405	[1][1]
Supplier	900		900		405	[1][1]

Table 8.11: Desired output 3rd comparison EC case.

	Current MR		New MR		Address	
Stakeholder	Value	Write Seq	Value	Write Seq	Value	Write Seq
EMP	105	[1][2]	107	[1][2][1]	409	[1][2][1]
GridOperator	200		200		409	[1][2][1]
MRParty	300		300		409	[1][2][1]
NewMRParty	105	[1][2]	107	[1][2][1]	409	[1][2][1]
OldMRParty	105	[1][2]	107	[1][2][1]	500	
OldPVShipper	105	[1][2]	107	[1][2][1]	600	
OldSupplier	105	[1][2]	107	[1][2][1]	701	[2]
PVShipper	800		800		409	[1][2][1]
Supplier	900		900		409	[1][2][1]

Table 8.12: Erroneous output 3rd comparison EC case.

TC - First comparison: Buy Packages and Options – Close Customer without Freezing

In this comparison, the situation is analyzed where a customer creates new orders on packages and options while his account is closed at about the same time. Three datafields are traced: *AccessADSL*, *AccessDialUp*, and *AccessWiFi*, which hold the status of the services available to the customer. The initial values of the first comparison are provided in Table 8.13.

In Table 8.15, an example is provided of the erroneous output represented by Table 4.9 in Chapter 4.

System	AccessADSL	AccessDialUp	AccessWiFi
BPM Layer	100	100	100
Prov. Interface	300	300	300
Front End	500	500	500
Infranet	700	700	700

Table 8.13: Initial values comparison 1 TC.

	AccessADSL		AccessDialUp		AccessWiFi	
Stakeholder	Value	Write Seq	Value	Write Seq	Value	Write Seq
BPM Layer	116	[1][1][2][2][2][2]	117	[1][1][2][2][2][2]	118	[1][1][2][2][2][2]
Prov. Interface	116	[1][1][2][2][2][2]	117	[1][1][2][2][2][2]	118	[1][1][2][2][2][2]
Front End	500		500		500	
Infranet	700		700		700	

Table 8.14: Desired output 1st comparison TC.

	AccessADSL		AccessDialUp		AccessWiFi	
Stakeholder	Value	Write Seq	Value	Write Seq	Value	Write Seq
BPM Layer	316	[2][2][2][2][1][1]	317	[2][2][2][2][1][1]	318	[2][2][2][2][1][1]
Prov. Interface	316	[2][2][2][2][1][1]	317	[2][2][2][2][1][1]	318	[2][2][2][2][1][1]
Front End	500		500		500	
Infranet	700		700		700	

Table 8.15: Erroneous output 1st comparison TC.

TC - Second comparison: Customer Move – Close Customer at End of Contract Terms

In this comparison, the situation is analyzed where a customer decides to move to a new address while his account is closed at about the same time. Three datafields are traced: *CustomerBlocking*, *Services* and *Address*. The initial values of the second comparison are provided in Table 8.16.

The desired outcome is obtained by executing Customer Move and Close customer sequentially. That is, first the customer moves, next the contract is ended. The output is shown in Table 8.17.

In Table 8.18, an example is provided of the erroneous output represented by Table 4.10 in Chapter 4.

System	CustomerBlocking	Services	Address
BPM Layer	100	100	100
Prov. Interface	300	300	300
Front End	500	500	500
Infranet	700	700	700

Table 8.16: Initial values comparison 2 TC.

	CustomerBlocking		Services		Address	
Stakeholder	Value	Write Seq	Value	Write Seq	Value	Write Seq
BPM Layer	711	[1][2][2][2]	704	[1]	702	[1]
Prov. Interface	310	[2][2][2][2]	300		300	
Front End	500		500		500	
Infranet	711	[1][2][2][2]	704	[1]	702	[1]

Table 8.17: Desired output 2nd comparison TC.

	CustomerBlocking		Services		Address	
Stakeholder	Value	Write Seq	Value	Write Seq	Value	Write Seq
BPM Layer	110	[2][1][2][2]	111	[1]	703	[1]
Prov. Interface	308	[2][2][2][2]	300		300	
Front End	500		500		500	
Infranet	110	[2][1][2][2]	111	[1]	703	[1]

Table 8.18: Erroneous output 2nd comparison TC.

TC - Third comparison: Upgrade/Downgrade/Switch – Upgrade from ADSL to VOIP/IPTV/Broadband

In this comparison, the situation is analyzed where two processes are executed simultaneously to update a package of products. The first process is using the gathered data to downgrade a package, whereas the second is using the same data for upgrading the package to a Broadband package. The datafield *AccountDetails* is traced. The initial values of the third comparison are provided in Table 8.19.

System	**AccountDetails**
BPM Layer	100
Prov. Interface	300
Front End	500
Infranet	700

Table 8.19: Initial values 3rd comparison TC.

The desired outcome is obtained by executing *Upgrade/Downgrade/Switch* and *Upgrade to Broadband* sequentially. That is, first the package is downgraded, next the package is upgraded to broadband. The output is shown in Table 8.17.

	AccountDetails	
Stakeholder	Value	Write Seq
BPM Layer	704	[1][1][2]
Prov. Interface	300	
Front End	704	[1][1][2]
Infranet	703	[1][1]

Table 8.20: Desired output 3rd comparison TC.

In Table 8.21, an example is provided of the erroneous output represented by Table 4.11 in Chapter 4.

	AccountDetails	
Stakeholder	Value	Write Seq
BPM Layer	704	[1][1]
Prov. Interface	300	
Front End	102	[1][2]
Infranet	704	[1][1]

Table 8.21: Erroneous output 3rd comparison TC.

B. BP represention of the WMO process

For brevity and clarity reasons, aliases are used instead of the full activity or variable identifiers, i.e. the complete references to service invocation methods, parameters and state variables which reside in the SR. For instance, the full identifier $TenderWCSupplier.12CB.tenderDecision$ is represented by the activity $decision$. Moreover, we have omitted the declaration of the local process variables that are used for storing the outputs of activities (e.g. $tmp\_hvOut\_homeInfo$).

```
<BusinessProcess name="WMO">
<input>
  <parameter name="bpAddress" type="dt:address"/>
  <parameter name="bpCid" type="dt:citInfo"/>
  <parameter name="bpEligCrit" type="dt:lawInfo"/>
  <parameter name="bpMedCond" type="dt:medInfo"/>
</input>

<sequence>
  <execute name="intake"
        input="itIn_Cid=bpCid;itIn_address=bpAddress"
        output="tmp_itOut_prov:=itOut_prov"/>

  <repeatUntil>
    <sequence>
      <execute name="homeVisit"
               input="hvIn_Cid=bpCid;hvIn_address=bpAddress"
               output="tmp_hvOut_homeInfo:=hvOut_homeInfo;tmp_hvOut_maRequired:=
                    hvOut_maRequired"/>

      <DS name="DS0">
        <guard>
          <variables>
            <variable name="bpAddress"/>
            <variable name="bpMedCond"/>
          </variables>
          <sequence>
            <switch>
              <case condition="hvOut_maRequired=true">
                <execute name="medicalAdvice"
                         input="maIn_cid=bpCid"
                         output="tmp_maOut_medInfo:=maOut_medInfo"/>
              </case>
              <otherwise>
                <empty/>
              </otherwise>
            </switch>
        <execute name="Decision"
                 input="dcIn_cid=bpCid;dcIn_homeInfo=tmp_hvOut_homeInfo;
                     dcIn_eligCrit=bpEligCrit;dcIn_medInfo=tmp_maOut_medInfo"
                 output="tmp_dcOut_approvalCheck:=dcOut_approvalCheck"/>
      </sequence>
    </guard>
    <verify>
      <case condition="bpAddress.county!='Groningen'">
        <terminate>
          <achieve-maint>
            <eq-val var="notifiedCityHall" value="TRUE"/>
            <eq-val var="messagePar" value="countyChange"/>
          </achieve-maint>
        </terminate>
      </case>
      <case condition="bpAddress.county='Groningen'">
        <achieve-maint>
          <known var="dcOut_approvalCheck"/>
        </achieve-maint>
      </case>
    </verify>
  </DS>

  <switch name="rejected">
    <case condition="dcOut_approved=false">
      <pick>
        <onMessage variable="appeal">
          <switch name="appealGranted">
           <case condition="appeal='granted'">
            <empty/>
           </case>
           <otherwise>
            <exit/>
           </otherwise>
          </switch>
        </onMessage>
        <onAlarm><for>'PT14D'</for>
          <exit/>
        </onAlarm>
      </pick>
    </case>
    <otherwise>
      <empty/>
    </otherwise>
  </switch>

</sequence>
<condition>dcOut_approved=true</condition>
```

```
</repeatUntil>

<switch name="selectProvision">
  <case condition="itOut_prov='care␣in␣kind'">
    <sequence>
      <DS name="DS3">
        <guard>
          <variables>
            <variable name="bpAddress"/>
            <variable name="bpMedCond"/>
           </variables>
          <sequence>
            <execute name="sendOrder"
                     input="sdhrIn_cid=bpCid;sdhrIn_orderInfo=
                          tmp_hvOut_homeInfo;sdhrIn_address;bpAddress"
                     output="orderId:=sdhrOut_orderId;orderContents:=
                          sdhrIn_orderInfo"/>
            <execute name="receiveDeliveryConfirmation"
                     input="dlIn_cid=bpCid;dlIn_id=orderId;dlIn_address=
                          bpAddress;dlIn_delContents=orderContents"
                     output="tmp_dlOut_conf:=dlOut_conf"/>
          </sequence>
        </guard>
        <verify>
          <case condition="bpAddress.county!='Groningen'">
            <terminate>
              <achieve-maint>
                <eq-val var="notifiedCityHall" value="TRUE"/>
                <eq-val var="messagePar" value="countyChange"/>
                <invalid var="orderId"/>
              </achieve-maint>
            </terminate>
          </case>
          <case condition="bpAddress.county='Groningen'␣AND␣bpMedCond!='
               deceased'">
            <achieve-maint>
              <known var="tmp_dlOut_conf"/>
            </achieve-maint>
          </case>
          <case condition="bpMedCond='deceased'">
            <terminate>
              <achieve-maint>
                <invalid var="orderId"/>
              </achieve-maint>
            </terminate>
          </case>
        </verify>
      </DS>
    <execute name="handleInvoice"
             input="hiIn_cid=bpCid;riIn_id=orderId"
             output="tmp_hiOut_invId:=hiOut_invId"/>
  </sequence>
</case>
<case condition="itOut_prov='personal␣budget'">
  <empty/>
</case>
<otherwise>
  <sequence>
    <DS name="DS1">
      <guard>
        <variables>
          <variable name="bpAddress"/>
          <variable name="bpMedCond"/>
        </variables>
        <sequence>
          <switch>
            <case condition="itOut_prov='wheelchair'">
              <sequence>
                <execute name="acquireRequirements"
                         input="arIn_cid=bpCid;adIn_homeInfo=
                             tmp_hvout_homeInfo"
                         output="tmp_arOut_requirements:=arOut_requirements"/
                             >
                <execute name="sendOrder"
                         input="soIn_cid=bpCid;soIn_orderInfo=
                             tmp_arOut_requirements;soIn_address=bpAddress"
                         output="orderId:=soOut_orderId;orderContents:=
                             soIn_orderInfo"/>
              </sequence>
            </case>
            <case condition="itOut_prov='home␣modification'">
              <DS name="DS2">
                <guard>
                  <variables>
                     <variable name="bpEligCrit"/>
                   </variables>
                  <sequence>
                    <repeatUntil>
                      <execute name="tenderProcedure"
                               input="tpIn_cid=bpCid;tpIn_homeInfo=
                                   tmp_hvOut_homeInfo"
                               output="tmp_tpOut_tenderSelected:=
                                   tpOut_tenderSelected"/>
                      <execute name="checkTender"
                               input="ctIn_cid=bpCid;ctIn_selTender=
                                   tmp_tpOut_tenderSelected;ctIn_eligCrit=
                             bpEligCrit"
                         output="tmp_ctOut_tenderOK:=ctOut_tenderOK"/>
                <condition>ctOut_tenderOK=true</condition>
              </repeatUntil>
              <execute name="sendOrderConfirmation"
                       input="sosIn_cid=bpCid;sosIn_sid=
                           tmp_tpOut_tenderSelected;sosIn_orderInfo=
                           tmp_hvOut_homeInfo;sosIn_address=bpAddress"
                       output="orderId:=sosOut_orderId;orderContents:=
                           sosIn_orderInfo"/>
            </sequence>
          </guard>
          <verify>
            <achieve-maint>
              <known variable="orderId"/>
            </achieve-maint>
          </verify>
        </DS>
      </case>
    </switch>
    <execute name="receiveDeliveryConfirmation"
             input="dlIn_cid=bpCid;dlIn_id=orderId;dlIn_address=
                 bpAddress;dlIn_delContents=orderContents"
             output="tmp_dlOut_conf:=dlOut_conf"/>
  </sequence>
</guard>
<verify>
  <case condition="bpAddress.county!='Groningen'">
    <terminate>
      <achieve-maint>
        <eq-val var="notifiedCityHall" value="TRUE"/>
        <eq-val var="messagePar" value="countyChange"/>
        <invalid var="orderId"/>
      </achieve-maint>
    </terminate>
  </case>
  <case condition="bpAddress.county='Groningen'␣AND␣bpMedCond!='
      deceased'">
    <achieve-maint>
      <known variable="dlOut_conf"/>
    </achieve-maint>
  </case>
  <case condition="bpMedCond='deceased'">
    <terminate>
      <achieve-maint>
        <invalid variable="orderId"/>
      </achieve-maint>
    </terminate>
```

```
            </case>
          </verify>
        </DS>
        <execute name="handleInvoice"
                 input="hiIn_cid=bpCid;riIn_id=orderId"
                 output="tmp_hiOut_invId:=hiOut_invId"/>
      </sequence>
    </otherwise>
  </switch>
  <execute name="payment"
           input="pmIn_invId=tmp_hiOut_invId"
           output="tmp_pmOut_conf:=pmOut_conf"/>
</sequence>
</BusinessProcess>
```

C. Variable interdependencies

The variable interdependencies of the WMO process are can be defined as follows:

$dependsOn(bpAddress) = \{hvOut\_homeInfo\}$

$dependsOn(hvOut\_homeInfo) = \{maOut\_medInfo, dcOut\_approvalCheck,$

$arOut\_requirements, tpOut\_tenderSelection\}$

$dependsOn(tpOut\_tenderSelection) = \{ctOut\_tenderOK\}$

$dependsOn(bpMedCond) = \{maOut\_medInfo, dcOut\_approvalCheck,$

$arOut\_requirements, ctOut\_tenderOK\}$

$dependsOn(bpEligCrit) = \{ctOut\_tenderOK\}$

D. Modelling the WMO process as a planning domain

Intake(itIn_cid, itIn_address)

Prec:

$itIn\_cid = bpCid \ \wedge itIn\_address = bpAddress$

Eff:

$sense(itOut\_prov)$

HomeVisit(hvIn_cid, hvIn_address)

Prec:

$hvIn\_cid = bpCid \ \wedge \ hvIn\_address = bpAddress$

$known(itOut\_prov)$

Eff:

$sense(hvOut\_homeInfo) \ \wedge \ sense(hvOut\_maRequired)$

MedicalAdvice(maIn_cid)

Prec:

$maIn\_cid = bpCid \ \wedge \ known(hvOut\_maRequired) \wedge$

$hvOut\_maRequired = true \ \wedge \ known(hvOut\_homeInfo)$

Eff:

$sense(maOut\_medInfo)$

Decision(dcIn_cid, dcIn_homeInfo, dcIn_eligCrit, dcIn_medInfo)

Prec:

$dcIn\_homeInfo = hvOut\_homeInfo \ \wedge \ dcIn\_cid = bpCid \wedge$

$(\neg hvOut\_maRequired \ \vee \ known(maOut\_medInfo) \wedge$

$(hvOut\_maRequired \ \vee \ true) \ \wedge \ \neg known(dcOut\_approvalCheck) \wedge$

$(\neg hvOut\_maRequired \ \vee \ dcIn\_medInfo = maOut\_medInfo)$

Eff:

$sense(dcOut\_approvalCheck)$

AcquireRequirements(arIn_cid, arIn_homeInfo)

Prec:

$(itOut\_prov = 3 \ \vee \ itOut\_prov = 4) \ \wedge \ itOut\_prov = 3 \ \wedge$

$arIn\_cid = bpCid \wedge$

$arIn\_homeInfo = hvOut\_homeInfo \wedge$

$known(dcOut\_approvalCheck) \ \wedge \ dcOut\_approvalCheck = true$

Eff:

$sense(arOut\_requirements)$

TenderProcedure(tpIn_cid, tpIn_homeInfo)

Prec:

$(itOut\_prov = 3 \ \vee \ itOut\_prov = 4) \ \wedge \ itOut\_prov = 4) \wedge$

$tpIn\_cid = bpCid \ \wedge \ tpIn\_homeInfo = hvOut\_homeInfo \wedge$

$known(dcOut\_approvalCheck) \ \wedge \ dcOut\_approvalCheck = true$

Eff:

$sense(tpOut\_tenderSelected)$

CheckTender(ctIn_cid, ctIn_selTender, ctIn_eligCrit)

Prec:

$ctIn\_cid = bpCid \;\wedge\; ctIn\_selTender = tpOut\_tenderSelected, ctIn\_eligCrit = bpEligCrit$

Eff:

$sense(ctOut\_tenderOK) \wedge$

$(ctOut\_tenderOK = \mathit{false}) \;\Rightarrow\; invalidate(tpOut\_tenderSelection)$

SendOrder(soIn_cid, soIn_orderInfo, soIn_address)

Prec:

$soIn\_cid = bpCid \;\wedge\; soIn\_address = bpAddress \wedge$

$known(arOut\_requirements) \;\wedge\; soIn\_orderInfo = arOut\_requirements \wedge$

$\neg known(orderId)$

Eff:

$sense(soOut\_orderId) \;\wedge\; assign(orderId, soOut\_orderId) \wedge$

$assign(orderContents, soIn\_orderInfo)$

SendOrderToSelSupplier(sosIn_cid, sosIn_sid, sosIn_orderInfo, sosIn_address)

Prec:

$sosIn\_cid = bpCid \;\wedge\; sosIn\_sid = tpOut\_tenderSelected \wedge$

$known(ctOut\_tenderOK) \;\wedge\; ctOut\_tenderOK = \mathit{true} \wedge$

$sosIn\_address = bpAddress \;\wedge\; sosIn\_orderInfo = hvOut\_homeInfo \wedge$

$\neg known(orderId)$

Eff:

$sense(sosOut\_orderId) \;\wedge\; assign(orderId, sosOut\_orderId) \wedge$

$assign(orderContents, sosIn\_orderInfo)$

SendDHRequest(sdhrIn_cid, sdhrIn_orderInfo, sdhrIn_address)

Prec:

$(itOut\_prov = \mathit{1} \;\vee\; itOut\_prov = \mathit{2}) \;\wedge\; itOut\_prov = \mathit{2}) \wedge$

$sdhrIn\_cid = bpCid \;\wedge\; sdhrIn\_address = bpAddress \wedge$

$sdhrIn\_orderInfo = hvOut\_homeInfo \ \wedge \ known(dcOut\_approvalCheck) \wedge$

$dcOut\_approvalCheck = true \ \wedge \ \neg known(orderId)$

Eff:

$sense(sdhrOut\_orderId) \ \wedge \ assign(orderId, sdhrOut\_orderId) \wedge$

$assign(orderContents, sdhrIn\_orderInfo)$

DeliveryConfirmation(dlIn_cid, dlIn_id, dlIn_address, dlIn_delContents)

Prec:

$dlIn\_cid = bpCid \ \wedge \ dlIn\_id = orderId \wedge$

$dlIn\_delContents = orderContents$

Eff:

$sense(dlOut\_conf)$

ReceiveInvoice(riIn_cid, riIn_id)

Prec:

$riIn\_cid = bpCid \ \wedge \ riIn\_id = orderId \wedge$

$known(dlOut\_conf)$

Eff:

$sense(riOut\_invId)$

CheckInvoice(ciIn_invId)

Prec:

$known(riOut\_invId) \ \wedge \ ciIn\_invId = riOut\_invId \wedge$

$\neg known(ciOut\_invoiceOK)$

Eff:

$sense(ciOut\_invoiceOK)$

ReturnInvoice(rtiIn_invId)

Prec:

$known(riOut\_invId)\ \wedge\ riOut\_invId = rtiIn\_inveId$

$\wedge ciOut\_invoiceOK = false$

Eff:

$invalidate(riOut\_invId)\ \wedge\ invalidate(ciOut\_invoiceOK)$

Payment(pmIn_invId)

Prec:

$(\neg(itOut\_prov = 1\ \vee\ itOut\_prov = 2) \vee$

$((\neg itOut\_prov = 1\ \vee\ known(dcOut\_approvalCheck) \wedge$

$(\neg itOut\_prov = 2\ \vee\ known(ciOut\_invoiceOK))))$

$\wedge\ (\neg(itOut\_prov = 3\ \vee\ itOut\_prov = 4)\ \vee\ known(ciOut\_invoiceOK))$

$\wedge\ pmIn\_invId = riOut\_invId$

Eff:

$sense(pmOut\_conf)$

CancelOrder(coIn_orderId)

Prec:

$known(orderId)\ \wedge\ coIn\_orderId = orderId$

Eff:

$invalidate(orderId)$

NotifyCityHall(nchIn_msg)

Prec:

$\varnothing$

Eff:

$sense(nchOut\_sent)$

List of abbreviations

AA	Atomic Action
AAS	Atomic Action Set
ACID	Atomicity, Consistency, Isolation, Durability
AI	Artificial Intelligence
API	Application Programming Interface
ASV	Atomic Service Variables
BP	Business Process
BPAS	BP-specific Actions Set
BPEL	Business Process Execution Language
BPM	Business Process Management
BPMN	Business Process Modelling Notation
BPMP	Business Process Management Platform
BPPM	Business Process Pertinent Model
CS	Critical Section
CSP	Constraint Satisfaction Problem
CTL	Computation Tree Logic
DA	Dependent Activity
DE	Dependent Element
DG	Domain Generator
DS	Dependency Scope
DSRM	Design Science Research Methodology
EC	Energy Company

ECH	Energy Clearinghouse
EDSN	Energy Data Services Netherlands
EIS	Enterprise Information System
IOPE	Input, Output, Preconditions, Effects
IP	Intervention Process
IS	Information System
LS	Local Storage
MC	Measuring / Metering Company
OWL-S	Semantic Web Ontology Language
PD	Planning Domain
PE	Process Executor
PHCS	Process History Capture System
PI	Process Instance
PLM	Process Lifecycle Monitor
PM	Process Modeller
PN	Petri Net
PV	Process Variable
SI	Service Instance
SD	Service Description
SOA	Service-Oriented Architecture
SOC	Service-Oriented Computing
SR	Service Repository
ST	Service Type
TC	Telecom Company
TSO	Transmission System Operator
UML	Unified Modelling Language
VV	Volatile Variable
WfN	Workflow Net
WMO	Law of societal support (Wet Maatschappelijke Ondersteuning)
WSDL	Web Services Description Language
WSDL-S	Semantic Web Services Description Language
XML	eXtensible Markup Language

Acknowledgements

The process of writing a Ph.D. thesis is something which cannot be accomplished alone. In the past five years, I have been supported by many people, to whom I would like to express my gratitude in this way.

First and foremost, I would like to express my gratitude to my promotor, prof. dr. ir. Hans Wortmann, for giving me the opportunity to pursue a Ph.D. I enjoyed working with Hans and I am looking forward to continue our collaboration. I would also like to thank my co-promotor, dr. Alexander Lazovik, for the many detailed discussions regarding the technical part of the developed concepts.

I would like to thank the members of the reading committee, prof. dr. ir. W.M.P. van der Aalst, prof. dr. G.B. Huitema and prof. dr. M. Weske, for assessing this thesis and providing constructive and valuable comments.

Several people have contributed to this thesis. I have worked extensively with Eirini Kaldeli and Pavel Bulanov. Our collaboration has led to highly valuable discussions and the most important contributions of this thesis. I am very grateful for their effort on the AI planner and the prototype implementation. Furthermore, I would like to thank Doina Bucur for carefully reviewing the formal part of Chapter 4.

Besides being my paranymphs, Jan Braaksma and Boyana Petkova are also very good friends and colleagues. We had great times together during work and after work during drinks and many other non-research related events.

Laura Măruşter introduced me to the academic world, which I greatly appreciate. With her I wrote my first two publications. I would like to thank Nick Szirbik for the inspiring conversations about science and the digressions to his many great stories about history.

I would like to thank all my friends and colleagues for the drinks, the coffee and the great conversations about anything but research.

Furthermore, the valuable support provided by the secretary cannot be left unnoticed. In particular, I would like to express my gratitude to Irene Ravenhorst for the countless occasions where she made certain that important documents were reviewed on time. She always managed to squeeze me into the ever-busy schedule of Hans whenever necessary.

I would like to thank the people and companies, whom I cannot address by name, for their time and providing insight in their business processes. I am very grateful to Tjitte Bouma and Heinjo Rozing for the many hours they spent on transforming the raw data to readable process models.

Finally, I would like to express my thanks and gratitude to my parents Rob and Ria, my brother Marc and my sister Juliette who have always supported me and motivated me throughout my life. Without them, this thesis would definitely not be there.

Nick van Beest

Groningen, The Netherlands
January 2013

Summary

Business processes are found everywhere in modern organizations. Increasingly, business processes are supported by automated means. Concurrent execution of business processes is common in most organizations, though these processes may (partially) use the same resources in terms of information required. Such mutual dependency on process variables may cause inconsistencies during process execution, especially in highly distributed service environments. Although these processes may properly terminate, they may lead to undesirable outcomes from a business perspective. This is due to interference via changing data in process instances running concurrently. The situation where data is simultaneously modified by several processes is known as process interference.

Process interference occurs far more often than most people realize. Because there is often not an immediate software error, the incorrect impression exists that the process runs well. Nevertheless, in the real world these interferences lead to wrong invoices, wrong addresses, wrong decisions and so on. These errors in the real world lead to customer complaints, legal cases, and many untraceable societal costs but not to the root cause: the fact that process interference is not properly solved in process management software architecture.

In Chapter 4 of this thesis, the process interference problem is defined formally using temporal logic (CTL*). This formal specification provides the temporal characteristics of interference. Based on this formal definition, two case studies were conducted at a large Energy company and a large Telecom company in the Nether-

lands, to identify erroneous outcomes as a result of process interference. The process descriptions are based on detailed documentation about the process and user experience. Due to the complexity of the analysis, a software tool has been developed to simulate the different concurrent execution combinations. This tool provides the functionality to provide a complete overview of the erroneous situations. The analysis shows that process interference is far more than a rare unfortunate exception. It is widespread in these organizations and many interference cases could be identified.

In order to resolve this problem, a number of design concepts have been proposed and tested in Chapter 5 to prevent process interference by awareness of process dependencies and automatic execution of compensation activities. *Dependency scopes* are introduced to represent the dependencies between processes and data sources and mark the critical sections of the process that are vulnerable for interference. A dependency scope is a part of the business process with a set of volatile process variables, where the activities of the dependency scope are implicitly or explicitly relying on the accuracy of those process variables. *Intervention processes* are introduced to repair inconsistencies during execution of the process. An intervention process is a sub-process, comprising a set of compensation activities, which together restore the consistent state of a business process. These modeling concepts can be seamlessly integrated in existing Business Process Modeling platforms.

In Chapter 6, these concepts are further developed and automated. Consequently, the dependency scopes can be generated at design time based on the process model and the information available from the used software services. The intervention process is generated during runtime, when a change in the volatile process variables occurs. Based on a well-defined specification of the business process and declarative goals runtime inconsistencies can be resolved by employing AI planning. Both the dependency scope specification and the intervention process generation occurs automatically based on the existing business process specification. These solution concepts are powerful and do not require explicit modeling of all cases and conditions, but can be applied generically. Consequently, it is shown that process interference can be resolved during runtime without additional modelling effort for the process designer.

Finally, the performance and the feasibility have been tested. In order to evaluate the feasibility of the approach, an architecture has been designed and a prototype has been implemented in Chapter 7. The results indicate that coupling dependency scopes with declarative goals and generating intervention processes at runtime by means of AI planning is a usable and realistic method for resolving erroneous path situations caused by process interference. The proposed method is both sound and complete. The IP generated is finite in all cases. Although generated IPs may include finite loops through an enumerated repetition of certain activities, they cannot include indefinite loops, since a plan provided by the AI planner is a finite, partially ordered set of actions. Consequently, the generated intervention processes always satisfy the properties specified in the goal. If there exists a combination of activities that achieves the goal, then this sequence is found in the developed architecture.

Nederlandstalige samenvatting

In moderne organisaties worden bedrijfsprocessen veelal ondersteund door informatiesystemen. Parallelle uitvoer van bedrijfsprocessen komt frequent voor en deze processen kunnen (gedeeltelijk) gebruik maken van dezelfde informatie. Een dergelijke onderlinge afhankelijkheid tussen procesvariabelen kan inconsistenties veroorzaken in het proces, met name in sterk gedistribueerde service omgevingen. Hoewel deze processen uitgevoerd kunnen worden zonder softwarefouten, kunnen ongewenste resultaten optreden vanuit het perspectief van de klant. Dit wordt veroorzaakt door interferentie via datamutaties door processen die parallel worden uitgevoerd. De situatie waar gegevens gelijktijdig worden gemodificeerd door verschillende processen wordt procesinterferentie genoemd.

Procesinterferentie komt veel vaker voor dan men zich realiseert. In de meeste gevallen doet zich niet direct een onmiddellijke softwarefout, waardoor de verkeerde indruk kan bestaan dat het proces goed loopt. In werkelijkheid kunnen deze storingen echter leiden tot verkeerde facturen, verkeerde adressen, verkeerde beslissingen, etc. Dit resulteert in klachten van klanten, rechtszaken, en vele niet te traceren maatschappelijke kosten. De feitelijke oorzaak is echter lastig te traceren: het feit dat procesinterferentie niet goed wordt ondervangen in huidige informatiesystemen.

In hoofdstuk 4 van dit proefschrift wordt procesinterferentie formeel gedefinieerd met behulp van temporele logica (CTL*). Deze formele specificatie geeft de temporele karakteristieken van procesinterferentie weer. Op basis van deze formele specificatie worden twee case studies uitgevoerd, bij een groot energiebedrijf en een groot telecombedrijf in Nederland, om de foutieve resultaten te identificeren als gevolg van procesinterferentie. De procesbeschrijvingen zijn afkomstig van gedetailleerde documentatie over het proces en de ervaring van de gebruikers. Gezien de complexiteit van de analyse is een softwaretool ontwikkeld om verschillende situaties van parallelle uitvoer van processen te simuleren. Deze tool biedt de functionaliteit om een volledig overzicht te genereren van alle foutieve situaties. Deze analyse heeft aangetoond, dat procesinterferentie veel meer is dan een weinig voorkomende uitzondering in een goed lopend proces. Procesinterferentie komt vaak voor in deze organisaties en een groot aantal interferentie gevallen kon worden geïdentificeerd.

Om procesinterferentie te voorkomen is in hoofdstuk 5 een aantal modelleringsconcepten geïntroduceerd en getest. *Dependency scopes* zijn geïntroduceerd om de afhankelijkheden te representeren tussen processen en data en markeren de kritieke sectoren van het proces die gevoelig zijn voor interferentie. Een dependency scope is een deel van het bedrijfsproces met een set procesvariabelen, waar de activiteiten van de dependency scope impliciet of expliciet uitgaan van de juistheid van die procesvariablen. *Interventie processen* zijn geïntroduceerd om geconstateerde inconsistenties tijdens de uitvoering van het proces te repareren. Een interventie proces is een subproces, welke een reeks compensatie activiteiten omvat, die samen de consistente toestand van een bedrijfsproces herstellen. Deze modelleringsconcepten kunnen worden geïntegreerd in bestaande Business Process Modelling platformen.

In hoofdstuk 6 zijn deze concepten verder uitgewerkt en geautomatiseerd. Een algoritme is ontwikkeld, waarmee dependency scopes kunnen worden gegenereerd op basis van het procesmodel en de informatie die beschikbaar is van de gebruikte software services. De interventie processen worden tijdens de uitvoer van de processen gegenereerd, zodra er een verandering in de procesvariabelen is geconstateerd. Op basis van een goed gedefinieerde specificatie van het bedrijfsproces en de declaratieve doelen van die processes kunnen inconsistenties worden opgelost door het gebruik van KI planningstechnieken. Zowel de specificatie van de dependency scopes als de generatie van de interventie processen is automatisch op basis van de bestaande bedrijfsproces specificatie. Op deze manier is geen expliciete modellering vereist van alle specifieke situaties en omstandigheden. Procesinterferentie kan dus worden opgelost tijdens uitvoer van de processen zonder dat dit extra modellering vereist voor de procesontwerper, waardoor de ontwikkelde technieken generiek kunnen worden toegepast.

Ten slotte zijn de haalbaarheid van de methode en de prestaties getest. Om de haalbaarheid van de aanpak te evalueren, is een architectuur ontworpen en een prototype geïmplementeerd in hoofdstuk 7. Dit prototype is getest op een case van lokale overheden, waar ook de prestaties van de architectuur zijn gevalueerd. De resultaten laten zien dat het koppelen van dependency scopes met declaratieve doelen en het runtime genereren van interventie processen door middel van KI planningstechieken een bruikbare en realistische methode is voor het oplossen van inconsistenties als gevolg van procesinterferentie. De voorgestelde methode genereert interventie processen die zowel sound als volledig zijn. De interventie processen zijn eindig in alle gevallen. Hoewel de interventie processen lussen kunnen bevatten door middel van een herhaling van bepaalde activiteiten, zijn deze lussen altijd eindig, aangezien de planner per definitie een eindige reeks activiteiten genereert. De gegenereerde interventie processen voldoen in alle gevallen aan de eigenschappen die in het doel zijn gesteld. Als er een combinatie van activiteiten bestaat waarmee het doel kan worden bereikt, dan wordt in de ontwikkelde architectuur in alle gevallen een geschikt interventie proces gegenereerd.