Automated Runtime Repair of Business Processes

N.R.T.P. van Beest, E. Kaldeli, P. Bulanov, J.C. Wortmann, A. Lazovik

March 7, 2012

1 Introduction

Current organizations are characterized by long-running distributed business processes (BPs) involving many different stakeholders [21, 11]. The application of SOA in the field of Business Process Management enables the integration of interoperable, local or remote services within a BP, thus realizing complex composite functionalities, while aiming at adaptability and reuse. The adoption of a SOA perspective is particularly useful in business process domains such as the ones used in the field of e-Government, which are characterized by a large number of involved partners. In such a context, data resources used by the BP are not necessarily proprietary to the organization, but are shared with other stakeholders as well. As a result, data itself is highly distributed. Consequently, traditional verification techniques for workflow and data-flow (e.g. [30]) are not sufficient for ensuring the correctness of such BPs, as they assume that process and data interactions are predefined in advance. However, not all interactions are known or pre-specified, since data can be simultaneously accessed and modified by different processes, with no obvious relation to the BP in progress. As a result, inconsistencies occur during process execution, which may result in unexpected behavior and undesirable business outcomes. Often only end customers notice these problems, as they comprise erroneous orders or invoices, customer requests that are never handled, etc [5]. The situation where data is simultaneously modified by several processes is known as process interference [35, 4].

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. 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 [5], but not to the root cause: the fact that process interference is not properly solved in process management software architecture.
Most work about resolving process interference refers to failing processes or concerns design-time solutions [34, 31]. In [4], a run-time mechanism is provided, which uses dependency scopes and intervention processes to manage interferences discovered during execution. A dependency scope (DS) specifies a part of the BPs whose correct execution relies on the accuracy of a volatile process variable, i.e. a process variable that can be changed externally during the execution of the process. If a volatile variable is externally modified while the execution flow resides within the range of the respective DS, an intervention process (IP) is triggered as a response, with the purpose of resolving the potential inconsistencies stemming from this change event [4].

By using DSs, testing for unforeseen data interactions at each activity can be avoided. As a result, the process designer does not need to know all potential process interactions in advance. However, a significant amount of manual specification of the intervention patterns is required, since the appropriate IPs may differ considerably depending on the current execution state at which modification of a volatile variable occurred. For complex processes with numerous activities, it is very difficult and time-consuming to define IPs at design-time, as the amount of potential IPs may be particularly high. In addition to that, it cannot be ensured that all important intervention cases are taken into account. Moreover, as the same BP may be deployed and used by more than one organization, different intervention processes have to be specified for each potential interference case at each organization.

The workload due to extensive manual configuration can be significantly reduced by automating the task of IP generation. In this paper, domain-independent AI planning is employed to automatically generate IPs, which restore the consistency of a BP. In that way, the manual work required by the domain designer is reduced to the specification of a high-level goal, which describes in a declarative way the desired consistent state that has to be reached in case of interference. To realize such a level of automation, additional semantic annotations are required, which capture the functional aspects of the activities participating in the business domain in terms of preconditions and effects, in spirit with existing process ontologies such as OWL-S(...). However, as is shown later on, the semantic annotations reflecting the BP-specific restrictions and relations between the activities can be derived in an automatic way.

More specifically, the contributions of the current paper include a well-defined specification of a BP using dependency scopes and declarative goals for resolving runtime inconsistencies by employing AI planning, as well as a correspondence between the BP specification and a planning domain, along with an algorithm for automatically deriving the latter from the former. By examining a realistic case-study from the Dutch e-government, we show how the generation of intervention processes can be realized by state-of-the-art planning techniques. A fully-working prototype has been implemented, which builds upon the use of a domain-independent planner based on Constraint Satisfaction Problem (CSP) techniques [16]. Furthermore, it is equipped with a number of features that go beyond classical planning, and are of particular relevance to the requirements associated with BP modeling and repair. The framework has
been implemented and evaluated on performance, showing the feasibility of the approach. The focus of the work presented herein is to address process recovery from inconsistencies that result from process interference and are less obvious than the ones due to failing activities. However, the overall approach of using domain-independent AI planning for BP reconfiguration is more general, and can be used to react to any kind of events.

The remainder of this paper is organized as follows. Section 2 provides an overview of related work, and Section 3 describes a possible interference scenario on a real case-study taken from Dutch e-Government, which plays the role of our running example. The architecture of the proposed framework is described in Section 4. In Section 5 and 6, the definitions and methodologies regarding the proposed approach are presented. The implementation of the framework is described in Section 7. The performance of the implemented framework is evaluated in Section 8, and the overall conclusions are drawn in Section 9.

2 Related Work

The work presented herein shares many concerns with different subfields of BP management, including work in the areas of BP recovery, adaptation and process interference. Automated planning for the purpose of runtime BP reconfiguration has been proposed before in the literature, from different viewpoints, which are presented and compared in the followings, while research in planning techniques for semantic service composition is also relevant.

2.1 BP Changeability

Changeability of business processes is a large research area focusing on providing the capabilities to adapt business processes at design- or run-time and, thus, provide flexibility in information systems. As such, a number of well-known adaptability frameworks have been proposed. The most notable examples are the ADEPT project [6, 12], and the DECLARE framework [1]. 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 [6], of which an improved version has been proposed by [12]. The DECLARE framework utilizes the idea of a declarative process specification [1] in order to attain flexible process execution without losing support. 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 [33] 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 [26].

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 process inconsistencies. Consequently, requirements-driven changeability and adaptability can be considered orthogonal to our research.

In order to cover for 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 [24]. Similarly, existing runtime solutions for process interference are based on failing processes as well, e.g. [9, 34, 8]. That is, only those processes that fail during execution and terminate in an improper way are recovered. 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 [31]. 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. In [4], a runtime intervention approach is proposed to repair BPs upon interference. However, the design-time specification of the required IPs still requires an extensive manual effort. In order to automate the IP generation, some extra semantic annotations are required for describing the BP. The benefits of adding 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 [14], which rely on a description in terms of preconditions and effects, or process model generation [32].

2.2 Automated Planning for BP Reconfiguration

The advantages of integrating AI planning techniques for several applications in the field of Business Process Management have long been acknowledged. E.g. [28, 27, 22] focus on how different planning approaches can assist at the business process definition phase, while the work presented in [15] investigates how planning can be used 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. In [3] the use of an intelligent assistant based on AI planning techniques is discussed, 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.

In [7] the use of machine learning is proposed in order to infer the pre-
conditions and effects of activities provided the availability of a set of training examples, and then generate a partially ordered 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 our objectives are different, a common concern that we share with this framework’s approach is that we keep the BP-specific constraints decoupled from the generic service repository, thus allowing the planner to generate partially ordered plans with a high degree of flexibility.

The work closest to ours is the approach to BP adaptation through planning presented successively in [19, 20, 23]. This work uses several versions of Golog, which is based on planning by means of the situation calculus, to adapt a running process in case mismatches between the environment and the internal system representation are detected. In Golog the goal to be achieved has to be specified in a procedural way, as a non-deterministic program, as opposed to the use of high-level declarative goals, as the ones used by domain-independent planners, like the one presented herein. 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 in [23] is that it can manage any unforeseen event, by continually 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, and can thus not cover situations like the ones presented herein, which necessitate the fulfillment of extra requirements or the use of compensation activities. Moreover, no evaluation of the system is provided.

The approach adopted herein shares many common concerns with the work on semantic service composition by adopting AI planning methodologies [29, 17, 2], since the ultimate task is to combine actions in a dynamic way. 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, like in [29, 2]. Our approach of ad-hoc automatic process instance reconfiguration relies on a domain-independent planner that is presented in [17], where the domain designer just states what properties have to be satisfied, without having to anticipate how these can be fulfilled.

3 Process Interference: An Example

In order to illustrate the effects of process interference and the potential ways to overcome them, let us consider a real case-study from e-Government. This case-study concerns a BP with respect to the Dutch Law for Societal Support (known as the WMO law). 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 service-level and priorities may differ for each municipality, as the responsibility for the execution is defined locally at the municipality.

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.

3.1 WMO process description

The WMO process (shown in Figure 1) starts with the submission of an application for a provision by a citizen. 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.

3.2 Interference Examples

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.
Figure 1: WMO process model

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 2, 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.

Figure 2: Dependency scopes in the WMO process.
3.3 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 3a. However, if the order is already sent to the supplier, some additional activities are required (Figure 3b). 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.

Figure 3: Required intervention processes corresponding to DS1, in case of an address change

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 3c. Since the specifications on the order directly rely on the address, a change of address implies a cancellation of the order in all cases, if an order has already been sent. The remainder of the IP is identical, as shown in Figure 3d. 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 3e).

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 whenever possible, based on the DS, the current state and the value of the volatile process variable. In the next section, the architecture of the framework supporting IP generation is presented.

4 Architectural Overview

Figure 4 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, XOR etc., and design tools for modeling DSs, in accordance with their definition provided in Section 5. 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.

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 description, which specifies the interface of the service annotated by some extra semantics. These semantics represent each service operation 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 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 methods 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).

The Planning Domain is computed by the Domain Generator (DG) only once per BP, 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 Set (AAS) and the BP specification (provided as output by the PM) to the DG. The AAS represents the BP-pertinent action descriptions as kept in the SR (i.e., the ones referenced by the BP along with the compensation operations). More details about the formation of the AAS is provided in Section 6.1. Given these two inputs, the DG can generate the encoding of the Planning Domain (see Definition 4 in Section 5.3), by enriching the generic action descriptions in the AAS with extra preconditions and effects (using Algorithm 1) that reflect the BP-specific interdependencies between the actions (e.g., sequence, flow, XORs).

Given the Planning Domain, the initial state and the goal, the AI planner (which is based on constraint solving) 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 3a-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 3e). 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 2, 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, i.e., variable modifications that are received during the execution of an IP are covered by the same set of DSs that covered the action before which the planner was fired. 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 implementation of the PE and the PM is presented in Section 7, while more details about the AI planner are provided in Section 5.3 and Section 6.

5 Basic Concepts

In this section, the definition of the basic concepts is provided, where the approach for BP repair is built upon.

5.1 Business Process

First, we define the SR, which comprises a set of service descriptions and a set of service instances, which “implement” some service description. The service descriptions comprise semantics, which specify the functionality provided by a service type. The service instances specify the way to invoke a certain service conforming to a service description. The semantic markups defined in the service description 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 activity’s execution. The service 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\(^2\).

**Definition 1 (Service Repository (SR)).** A Service Repository \( SR = (SD, SI) \) is a storage, which keeps a set of Service Descriptions \( SD \), and a set of Service Instances \( SI \). A Service Description \( sd \in SD \) is a tuple \( sd = (sid, O, SV) \), where \( sid \) is a unique identifier, \( O \) is a set of service operations (or methods), and \( SV \) is a list of variables (alternatively called fields), each ranging over a finite domain. These variables correspond to state variables internal to the service, whose value can be changed by the methods of the service. Each service operation \( o \in O \) is a tuple \( o = (id(o), in(o), out(o), prec(o), eff(o)) \) 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 \( eff(o) \) a set of effects, as defined in Definition 4 with \( Var = in(o) \cup out(o) \cup SV \)

A Service Instance \( si \in SI \) is a tuple \( si = (iid(si), st(si)) \), where:

- \( st(si) \) is the unique identifier (service type) of the service description \( sd \in SD \) this instance complies with
- \( iid(si) \) is an instance identifier. For each pair of service instances \( si_1, si_2 \in SI \) that have the same service type \( st(si_1) = st(si_2) \), \( iid(si_1) \neq iid(si_2) \).

The SR plays the role of a pool service descriptions and instances, which are used as the building elements of different process specifications. In the followings, the definition of a Business Process (BP) is provided, which includes the basic activities and control structures such as sequence, flow and XOR. The BP is enriched with DSs, which also constitute parts of the process. The syntax of the BP is well-defined and unambiguous, so that they can be directly executed by the Process Executor (see Section 7.2) and automatically transformed to a representation usable by the planner. The BP definition used in this paper is block-structured, in spirit with BPEL’s notation. As shown in [25, 18], a BP representation following a graph-based model, such as the BPMN-like notation used in Figure 1, can be easily mapped to BPEL-like block structures, similar to the ones used in our definition. The representation we use herein constitutes

\(^1\)www.w3.org/Submission/WSDL-S
\(^2\)www.w3.org/Submission/OWL-S
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 2 (Business Process (BP)).** Given a Service Repository $SR = (SD, SI)$, a Business process is a tuple $BP = (PV, E)$, with $E$ being a process element $E = (act \mid seq \mid flow \mid XOR \mid repeat \mid while \mid DS)$, 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 declared at the BP level (BP-specific). A subset of these variables are passed as input parameters to the entire BP, in which case we write $BP(pv_1, \ldots, pv_n)$, where $pv_i \in PV_i$ and $pv_i$ can be initialized with specific values at execution time.
  - $PV_e$ is a set of external variables, which refer to variables declared in the $SR$. An external variable $v \in PV_e$ is a reference $sn.iid.fid$, with $sn$ being the identifier of a service description $sd = (sn, O, SV) \in SD$, $iid$ the identifier of a service instance $si = (iid, sn) \in SI$, and $fid$ the identifier of some state variable (field) $f \in SV$.

- $act$ is a process activity, which represents an invocation method that exists in $SI$. It is a tuple $act = (id(act), in(act), out(act))$, where $id(act)$ is a reference $sn.iid.oid$, where $sn$ is the identifier of a service description $sd = (sn, O, SV) \in SD$, $iid$ is the identifier of a service instance $si = (iid, st) \in SI$, and $oid$ is the identifier of some operation $o \in O$. The input and output parameters of $act$ refer to the input and output parameters of the respective $oid$, i.e. $in(act) = in(oid)$ and $out(act) = out(oid)$. 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. We thus write an action invocation as $id(act)(ip_1 := v_1, \ldots, ip_n := v_n)$, where $ip_i \in in(act), v_i \in (PV \cup OP)$, or $v_i$ is a value compliant with $ip_i$’s domain. There are also two extra special types of activities: no-op, which represents an idle activity with true preconditions and no effects, and terminate, whose execution causes the whole BP to halt. Directly after an action invocation, an action’s output can be stored in some process variable, in which case we write $(pv_1 := op_1, \ldots, pv_n := op_n)$, where $op_i \in out(a)$ and $pv \in PV_i$.

- $seq$ refers to a totally ordered set of process elements, which are executed in sequence. The following notation is used: $seq\{e_1 \ldots e_n\}$, where $e_i$ is a process element.

- $flow$ represents a set of process elements, which are executed in parallel. We write $flow\{e_1 \ldots e_n\}$, where $e_i$ is a process element.
**XOR** is a set of tuples \( \{ (c_1, e_1), \ldots, (c_n, e_n) \} \), where \( e_i \) is a process element and \( c_i \) is a logical condition \( C \), which conforms to the following syntax:

\[
C \ ::= \text{prop} \mid \lor C_j \mid \land C_j \mid \neg C_j
\]

\[
\text{prop} \ ::= \text{var} \circ \text{value} \mid \text{var1} \circ \text{var2} \mid (\text{var1} \circ \text{var2}) \circ \text{value},
\]

where \( \text{var}, \text{var1}, \text{var2} \in (\text{PV} \cup \text{OP}) \), \( \text{value} \) is some constant belonging to \( \text{var} \)'s domain, \( \circ \) is a relational operator (\( \circ \in \{=, <, >, \neq, \leq, \geq\} \)) and \( \circ \) a binary operator (\( \circ \in \{+,-\} \)). All \( c_i \) participating in an XOR are mutually exclusive, i.e. for any given assignment to \( \text{PV} \cup \text{OP} \), only a single \( c_i \) evaluates to true, and \( e_i \) will be executed if \( c_i \) evaluates to true. We write \( \text{XOR}\{c_1 \Rightarrow e_1, \ldots, c_n \Rightarrow e_n\} \).

**repeat** represents a loop structure, and is defined as a tuple \( (\text{pe}, c) \{ \text{pe}_i \} \), where \( c \) is a logical condition as already defined, and \( \text{pe}, \text{pe}_i \) are process elements. \( c \) is evaluated just after the end of \( \text{pe} \), and if it holds then \( \text{pe} \) is repeated, after the execution of the optional \( \text{pe}_i \). We write \( \text{repeat}\{\text{pe}\} \text{while}(c) \{ \text{pe}_i \} \), with \( \{ \text{pe}_i \} \) being optional.

**while** is similar to **repeat**, with \( c \) being evaluated before \( \text{pe} \) starts.

- **S** is a dependency scope as defined in Definition 3.

### 5.2 Dependency scope

The DS is based on a **guard-verify** structure analogous to the **try-catch** approach used in many programming languages to deal with exogenous events, and which is also used by [19]. 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 3** (Dependency Scope (DS)). Given a \( \text{SR} = (\text{SD}, \text{SI}) \) and a \( \text{BP} = (\text{PV}_i \cup \text{PV}_e, E) \), a dependency scope is a tuple

\[
\text{DS} = \langle \text{guard}(VV)\{E_g\}, \text{verify}((\{\text{case}(C_i) : G_i \mid E_{ip} \mid \text{terminate}(G_i) \mid \text{terminate}(E_{ip})\}))\rangle,
\]

where:

- \( \text{guard}(VV) \) indicates the set of volatile variables \( VV \subset \text{PV}_e \) whose modification triggers the verification of the DS, and \( E_g \) a process element in the \( \text{BP} \). Whenever during the execution of \( E_g \) an event indicating a change in the value of a volatile variable \( w \in VV \) is received, the **verify** part of the DS is triggered, and \( \text{BP} \)'s execution is interrupted.

- \( \text{verify}((\{\text{case}(C_i) : G_i \mid E_{ip}\})) \) 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, as defined in Definition 2. Providing a case condition is optional, with the default interpretation being \( C_i = \text{TRUE} \).

- \( G_i \) 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 [16]. After interrupting the BP execution, the plan that satisfies the respective \( G_i \) (if it can be found) is executed. When the plan’s execution is completed, the \( BP \) is resumed at the state after \( E_g \) 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 \( BP \)’s execution is interrupted, \( E_{ip} \) is executed, and after its completion \( BP \) resumes from the end of \( E_g \).

- \( \text{terminate}(G_i) \) (\( \text{terminate}(E_{ip}) \)) forces the process to terminate, i.e. abort the rest of \( BP \)’s execution, after fulfilling \( G_i \) (completing \( E_{ip} \)’s execution).

The complete specification of the full WMO BP, annotated with all DSs, is provided in Appendix A.1. Following Definition 3, the DS specification representing \( DS_1 \) of Figure 2 is the following:

```
guard(address, medCond){
  seq{
    XOR{
      /* subprocess */
    } receiveDeliveryConf(dlIn orderId=orderId, dlIn cid=bpCid, address=orderAddress, dlIn delContents=orderContents)
  }
}

verify{
  address.county \neq 'Groningen': \text{terminate(achieve-maint (notifiedCityHall('countyChange')=\text{TRUE} \& invalid(orderId)))}
  address.county = 'Groningen' \& \& medCond \neq deceased:
    achieve-maint(known(dlOut conf))
  medCond='deceased': \text{terminate(achieve-maint(invalid(orderId)))}
}
```

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 to another 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 set by “Intake and Application”)
has been canceled, and a respective notification is sent to the city hall. The plan will be equivalent to IP (e) of Figure 3.

- If the medical condition has changed to some new value that does not indicate “deceased” and the customer is still within the range of the municipality, 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 3. After the plan’s execution 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 3, 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 2. 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 can also be defined on top of the BP specification, prescribing the direct dependency of some variables on the validity of some other variable. The \( \text{dependsOn} \) relation is used for this purpose: \( \text{dependsOn}(v) = \{v_1, \ldots, v_n\} \). 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, \( \text{dependsOn}(\text{bpAddress_address}) = \{\text{hvOut_homeInfo}\} \), since \( \text{hvOut_homeInfo} \) refers to the information retrieved for the specific \( \text{hvIn_address} \). Thus, if the person moves to some other address, the collected information is not valid anymore. In turn, a set of variables, like \( \text{arOut_requirements} \) reflecting the acquired requirements concerning the wheelchair, are directly dependent on \( \text{hvOut_homeInfo} \). On the other hand, an \( \text{orderId} \) 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. The full set of variable interdependencies that accompany the WMO
BP specifications are provided in Appendix A.1.

5.3 The Planning Domain

The PE constructs a planning domain given a BP specification and a SR, which is used by the planner for generating the IPs upon recovery requests. In this subsection, the definition of a Planning Domain (PD) is provided, in line with [17]. A step-by-step explanation of the automatic composition of the PD is given in Section 6.

**Definition 4 (Planning Domain (PD)).** A Planning Domain is a tuple \( PD = \langle Var, Par, A \rangle \), 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 triple \( a = (id(a), in(a), precond(a), effects(a)) \), 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:
    \[
    \text{precond}(a) ::= \text{prop} \land_i \text{precond}(a) \lor_i \text{precond}(a) \neg \text{precond}(a)
    \]
    \[
    \text{prop} ::= \text{var} \circ val \mid \text{var1} \circ \text{var2} \circ val \mid \text{known}(\text{var}),
    \]
    where \( \text{var}, \text{var1}, \text{var2} \in (Var \cup Par) \), \( val \) is some constant, and \( \circ \) is a relational operator (\( \circ \in \{=, <, >, \neq, \leq, \geq\} \)) and \( \circ \) a binary operator (\( \circ \in \{+,-\} \)).
  - \( effect(a) \) is a conjunction of any of the following elements:
    * \( \text{assign}(\text{var}, v) \), where \( v \) is some constant or \( v \in Var \)
    * \( \text{assign}(\text{var}, f(v_1, v_2)) \), where \( v_1, v_2 \in (Var \cup Par) \) or \( v_1, v_2 \) are constants, and \( f \) the sum or the subtract function
    * \( \text{increase}(\text{var}, v) \) or \( \text{decrease}(\text{var}, v) \), where \( v \in Var \) or \( v \) is some constant
    * \( \text{sense}(\text{var}) \), where \( \text{var} \in Var \).
    * \( \text{invalidate}(\text{var}) \), where \( \text{var} \in Var \). This effect states that \( \text{var} \) becomes unknown.
    * \( \text{prop}(a) \Rightarrow effect(a) \), 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 \( s = (x_1, D_s^{x_1}), \ldots, (x_n, D_s^{x_n}) \), where \( x_i \in \text{Var} \cup \text{Par} \) 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 \( x(s) = D_s^{x_i} \) if \((x, D_s^{x_i}) \in s\). If \(|D_s^{x_i}| = 1\), this means that \( x \) at \( s \) has a specific value. The domain modelling is based on the Multi-valued Planning Task encoding [13], which leads to a smaller number of variables ranging over larger domains, and is particularly well-suited for constraint solvers. The effects of type \textit{sense} are called \textit{observational}, i.e. they observe the current value of a variable, while the \textit{assign} and \textit{increase/decrease} types of effects are \textit{world-altering}, i.e. actively change the value of a variable. An action may have both observational and world-altering effects.

Sensing effects model non-deterministic assignments to variables, whose value is unknown offline. For example, the result of the \textit{deferred choice} represented by the “ Decision” action in Figure 1 (i.e. \texttt{dcOut	extunderscore confirm}=true or \texttt{dcOut	extunderscore confirm}=false) is modelled via an effect of the form \textit{sense}(dcOut	extunderscore confirm). Sensing outcomes are commonly used in \textit{deferred choices}, i.e. XOR-splits 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 \textit{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 \texttt{cancelOrder(orderId)} has as an \textit{invalidate}(orderId), which entails that the \texttt{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 process. As a result, the repetition of the “Tender Procedure” is enforced for the process to go on. This behaviour is modelled by the effect \( \neg\text{tsOut	extunderscore tenderSelOK} \Rightarrow \text{invalidate}(c\text{tOut	extunderscore tenderSel}) \), and is automatically generated given the repeat structure of the BP specification, as it is explained in Section 6.

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 \( \text{var} \in \text{Var} \), a new boolean variable \( \text{var}\textunderscore known \) is introduced, which indicates whether \( \text{var} \) is known at state \( s \) (\( \text{var}\textunderscore known(s) = \text{true} \)) or not (\( \text{var}\textunderscore known(s) = \text{false} \)). Given these extra knowledge-level variables, \( \text{known}(\text{var}) \) is equivalent to stating \( \text{var}\textunderscore known = \text{TRUE} \). Similarly, \( \text{invalidate}(\text{var}) \) is equivalent to stating \( \text{assign}(\text{var}\textunderscore known, \text{FALSE}) \). For every variable \( \text{kvar} \in \text{Var} \) that participates in an observational effect, a new variable is introduced \( \text{kvar}\textunderscore response \), which is a placeholder for the value returned by the respective sensing operation. Since this value is unknown until execution time, \( \text{kvar}\textunderscore response \) ranges over \( \text{kvar} \text{’s} \) domain (\( \text{kvar}\textunderscore response} \in D^{\text{kvar}} \)). Thus,
sense(kvar) is equivalent to assign(kvar, kvar_response). Furthermore, for each variable cvar ∈ 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, 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.

5.4 Encoding the domain into a CSP

A constraint satisfaction problem is a triple \( CSP = \langle X, D, C \rangle \), where \( X = \{ x_1, \ldots, x_n \} \) is a finite set of \( n \) variables, \( D = \{ D^1, \ldots, D^n \} \) is the set of finite domains of the variables in \( X \) so that \( x_i \in D^i \), and \( C = \{ c_1, \ldots, c_m \} \) 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 \( CSP \langle X, D, C \rangle \) 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 \( C \).

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 representation \( PD = \langle V, A \rangle \), the target is to encode \( PD \) into a \( CSP \langle X_{CSP}, D, C \rangle \). First, for each variable \( x \in V \) ranging over \( D^x \), and for each \( 0 \leq i \leq k \), 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 \( 0 \leq i \leq k-1 \) a boolean variable \( a[i] \) is defined. This way, the computed plan can include parallel actions, which may save time at execution. 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 [10]. Frame axiom constraints are also generated, which guarantee that variables cannot change between subsequent states, unless some action that affects them takes place, i.e. for every \( v \in Var - (Par \cup Rv) \) and for each \( 0 \leq i \leq k - 1 \) the constraint \( \land_j (actionAff(v)_j = 0) \Rightarrow v[i] = v[i + 1] \) is added, where \( actionAff(v)_j \) are the actions affecting \( v \).

6 Automatic Intervention Process Generation

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

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 \( st.iid.oid \in BPPM \) of a service instance \( si = (iid, st) \in SI \), 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 Set, as defined below.

**Definition 5 (Atomic Actions Set (AAS)).** Given a Service Repository \( SR = (SD, SI) \), a BP, and a set of BP-pertinent Methods BPPM, the Atomic Actions Set (AAS) is defined as follows:

- When the PE receives a request to execute the BP, a unique instance reference \( bp-iid \) is assigned.

- For each method \( bpo = st.iid.oid \in BPPM \), the service description \( sd = (sid, O, SV) \in SD \) with \( sid = st \) is found, and the operation \( o = (id(o), in(o), out(o), prec(o), eff(o)) \in O \) with \( id(o) = oid \) is retrieved.

- For each input parameter \( ip_i \in in(o) \), a new input variable is created for \( st.iid.oid.ip_i \), with name \( bp-iid.sd.iid.oid.ip_i \) 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 \( bp-iid.sd.iid.oid.op_i \), 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 \( eff(bpo) \) are generated, by substituting each input and output parameter with name \( v \) appearing in \( prec(o) \) and \( eff(o) \) by the reference \( bp-iid.st.iid.oid.v \). In case of a service state variable \( var \in SV \) with local name \( v \), the reference is substituted with the universal name \( st.iid.v \), which is BP independent. If \( st.iid.v \) has not been met before, the respective variable with name \( st.iid.v \) and with domain identical to \( var \) is created.
This way, for each $act = st.iid.oid \in BPPM$ the invocation method description tuple $imd = (bp-iid.st.iid.oid, in(act), out(act), prec(act), eff(act))$ is created by the PE. Each $imd$ is converted to a planning action (as defined in Definition 4) $a = (id(a) = (bp-iid.st.iid.oid, in(a)) = in(act)), prec(a) = prec(act), eff(a) = eff(act))$. These actions form the AAS. The set of the instance-level inputs and outputs of all $bpo \in BPPM$ form the AtomicInputs($AI$) and the AtomicOutputs($AO$) respectively, while the service state variables involved in the preconditions or effects of the service descriptions of all $bpo \in BPPM$ form the Atomic Service Variables ($ASV$).

The AAS together with the set of variables $AI, AO, ASV$ formed as described in the above definition, reflect only the atomic-level semantics of the actions. In the context of a certain BP, the universal action descriptions in the AAS 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.

6.2 Generation of the Planning Domain

The Domain Generator is responsible for transforming the AAS to a Planning Domain that comprises a process-specific representation of actions participating in the particular BP, which restricts their use according to the BP structure, as well as the compensation activities that are allowed to be used by the respective IPs. The DG is called only once by the PE, the first time it needs to call the AI Planner. 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 2 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 enforce the repetition of $firstAct(pe)$. For example, if the outcome of “Check tender with decision” is negative, the previous supplier selection made by the “Tender procedure” becomes directly invalid, and another tender has to be selected. 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, the set of atomic action descriptions AAS (which comprise the activities participating in the BP plus the allowed compensation actions). By parsing the BP, it constructs 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 AAS. The BP is treated as a tree (represented as an XML tree), where the root is the outer-most element in
Algorithm 1 Automatic addition of BP-specific preconditions and effects given a BP specification and a set of atomic actions AAS. The resulting action descriptions in AAS constitutes the Planning Domain.

procedure PD(BP, AAS)  
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 assignments
            (ip_i := v) = parseNextInput(e)
            addPrec(getAction(id(e), AAS), '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), AAS), 'assign(bpVar, eOut_v)')
        end while
        addPrec(getAction(id(e), AAS), seqPrec(prevElem(e), BP))
    case XOR{(c_1, e_1), . . . , (c_n, e_n)}:
        while hasNextBranch(e) do //parse all branches of the XOR
            (c_i, e_i) = getNextBranch(e) //prec for all actions at the beginning of XOR
            ∀a_i ∈ firstAct(e_i): addPrec(getAction(id(a_i), AAS), 'c_i')
        end while
    case repeat(pe, c):
        ∀a_i ∈ 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), AAS), 'c ⇒ ∧ a_j ∈ pe, o_k ∈ out(a_j) invalidate(o_k)')
    case otherwise: continue
end while
end procedure
Algorithm 2 Functions 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 ‘∧_{o_j \in \text{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 ¬c ∧ seqPrec(pe, BP)
case XOR\{(c_1, e_1), \ldots, (c_n, e_n)\}: //e_i of XOR-branch c_i is valid if c_i
  return ‘∧ \_i (¬c_i ∨ seqPrec(e_i, BP))’
case flow\{e_1, \ldots, e_n\}: //all parallel e_i:s are valid
  return ‘∧ \_i seqPrec(e_i)’
case empty:
  if PREVACT(e) ≠ ∅ then
    seqPrec(PREVACT(e, BP))
  else
    return true
  end if
end function

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 ∧ i ≠ 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
  if parent(e, BP)=∅ then  //if root
    return ∅
  else  //in all other cases, previous is the previous of the parent
    prevElem(parent(e, BP))
  end if
Algorithm 3 Auxiliary functions used for adding XOR and repeat conditions as preconditions.

```plaintext
function firstAct(e, BP): Set[Element] //Find the first action(s) of an element
    match type(e)
    case XOR = \{ (e_1, e_1), \ldots, (e_n, e_n) \}:
        return firstAct(e_1, BP) ∪ \ldots ∪ firstAct(e_n, BP)
    case repeat = \{ pe, c\{ pe_i \} \}:
        return firstAct(pe, BP) ∪ \ldots ∪ firstAct(e_n, BP)
    case flow\{ e_1, \ldots, e_n \}:
        return firstAct(e_1, BP) ∪ \ldots ∪ 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 XOR = \{ (e_1, e_1), \ldots, (e_n, e_n) \}:
        return lastAct(e_1, BP) ∪ \ldots ∪ lastAct(e_n, BP)
    case repeat = \{ pe, c\{ pe_i \} \}:
        return lastAct(pe, BP) ∪ \ldots ∪ lastAct(e_n, BP)
    case flow\{ e_1, \ldots, e_n \}:
        return lastAct(e_1, BP) ∪ \ldots ∪ lastAct(e_n, BP)
    case seq\{ e_1, \ldots, e_n \}:
        return lastAct(e_n, BP)
    case activity:
        return e
end function
```

the specification, and the leaves are the activities. For each 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 \( a \)'s preconditions. Next, possible assignments of the outputs of \( a \) to BP variables of the form \( bpVar := eOut_{\cdot \cdot} \) are parsed, and the respective assign effect is added to the effects of \( a \).

The preconditions enforcing \( a \)'s sequence relation with respect to its preceding process element \( e \), as computed by the \( \text{prevElem} \) function in Algorithm 2, are returned by the function \( \text{seqPrec} \) in Algorithm 2. These preconditions ensure that the appropriate preceding actions are executed prior to \( a \), depending on the type of \( e \). More specifically, \( \text{seqPrec} \) obtains the preconditions corresponding to all execution paths that may lead to \( a \), by finding the last action(s) of the respective execution paths, and the possible respective conditions on which this path is depending. The function \( \text{prevElem}(a, BP) \) returns either the previous element of \( a \) in a sequence relation if such one exists, or otherwise it recursively goes back 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 \( e = \text{prevElem}(a, BP) \) is an activity, the precondition states that the outputs of \( e \) have to be known. If \( e \) is a sequence, then \( \text{seqPrec} \) is computed on the last element in that sequence. In case of a repeat, \( \text{seqPrec} \) 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 to a’s execution. 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 XOR = \{(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, i.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.

After taking care of the sequence preconditions, Algorithm 1 proceeds with checking the case where the current element in the tree is of type XOR. In this situation, for each branch(c_i, e_i) of the XOR 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 \texttt{firstAct} in Algorithm 3. \texttt{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 = \texttt{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 Appendix A.2 the final planning domain representing the WMO BP, as produced by the application of Algorithm 1, is presented.

The outcome of the algorithm is a \textit{BP-specific Actions Set (BPAS)}, which is the original AAS enriched with the extra preconditions and effects. Together with the set of variables consisting of the variables \texttt{AI}, \texttt{AO}, \texttt{ASV} as described in Section 6.1 and the internal process variables \texttt{PV}_i declared in the BP, they constitute the planning domain considered by the planner. The BP-specific planning domain is thus defined as \( PD = (\texttt{Var}, \texttt{Par}, \texttt{Act}) \) (see Definition 4), with \( \texttt{Var} = \texttt{PV}_i \cup \texttt{AO} \cup \texttt{ASV} \), \( \texttt{Par} = \texttt{AI} \), and \( \texttt{Act} = \texttt{BPAS} \).

### 6.3 Composition 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 \textit{dependsOn} sets, these are enriched during execution of the BP by the PE: if an action comprising an assignment effect \texttt{assign}(v', v) or an increase(decrease) effect \texttt{increase}(v', v) (\texttt{decrease}(v', v)), has been executed, variable v' is added automatically to the \textit{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 \( \in \texttt{PV} \) 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).

- For each variable var for which no specific value has been acquired yet, the respective knowledge variable \texttt{known_var} is set to false at the initial state (\texttt{known_var}(0) = false).
• Given a change event on a volatile variable \( vv \), the interdependency rules are parsed. For each \( \text{var} \in \text{dependsOn}(vv) \) known \( \text{var}(0) = \text{false} \), indicating that the value of \( \text{var} \) as reflected by the current state of execution is not valid. The same is done recursively for each \( \text{var}' \in \text{dependsOn}(\text{var}) \), for all \( \text{var} \in \text{dependsOn}(vv) \).

6.4 Generating the IP

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).

However, the generated IP only includes sequence and flow control constructs. As a result, XOR-constructs, where the value of the condition is unknown off-line, cannot be expressed as a predefined sequence. Therefore, 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 3c, it generates the plan that corresponds to the assumption that the output of “HomeVisit” \( hvOut\_maRequired = \text{false} \), 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, i.e. 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 plan. Otherwise 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 \text{achieve-maint(known(delOut\_delId))} (see Section ??), in order to obtain the IP shown in Figure 3c (the input parameters are omitted for brevity):

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

Execute HomeVisit Output: \( hvOut\_maRequired = \text{true} \), constraint violation, re-plan

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

Execute MedicalAdvice Output: \( maOut\_medInfo = 'Document12A' \)
Execute Decision Output: \( dcOut\_approvalCheck = \text{true} \)
Execute TenderProcedure Output: \( tpOut\_tenderSelection = 'ACMFrizianConstructions' \)
Execute CheckTender Output: \( cOut\_tenderOK = \text{false} \), constraint violation, re-plan

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

27
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 methods are invoked as part of the IP.

## 7 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 4.

### 7.1 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 flows, XOR splits etc., with an added support for DS modelling. 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 7.2. 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 5 presents a screenshot of the PM, showing the graphical representation of DS1 of the WMO process from Figure 2. 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 2. This representation is passed to the PE for execution, as described in the next subsection.

### 7.2 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 2, and with the BP input parameters instantiated to specific values. The PE works in cooperation with the Service Repository as described in Definition 1. The details of Service Instances implementation are out of scope of this paper, and for the purposes of the testing presented in Section 8 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 $st.iid.oid$ specified by an activity in the BP specification, the
Figure 5: Screenshot of the Process Modeller.

PE retrieves the respective description kept in the SR. For example, the activity “SendOrder” in Figure 6 refers to “HomeModification.iid.sendOrderToSelSupplier”, which corresponds to the method “sendOrderToSelSupplier” of the “HomeModification” service description, and is provided by the service instance with identifier “WMO_hm_GR” (see Definition 1). The service description of “HomeModification” service type description which includes three operation descriptions, as well as the service instance (provider) “WMO_hm_GR” are kept in the SR, as shown in Figure 6. 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 6 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 AAS 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 \( bp-iid \) to the running instance, and constructs the AAS along with the instance-level inputs and outputs \( AI \cup AO \) (as described in Section 6.1), which are added to the LS. Each service state variable \( sv \in ASV \) (see Section 6.1) is added to the LS if it does not already exist. This way, state variables of the AAS 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 \( bp-iid.v \) 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 \( bp-iid \) 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

Figure 6: Example of a Service Description and a Service Instance.
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 3), 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”, and thus treats the modification events it receives during the IP execution in the same way as it did during execution of the process element covered by the DS in the BP. This means that an IP “inherits” the DSs that covered the activity before which the planner was invoked. In case a DS condition is triggered, the current IP execution is interrupted, a new IP is generated, after whose completion, 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 4). To this end, the PE calls the Domain Generator, by passing to it the Atomic Actions Set (AAS), built as described in Section 6.1 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 XORs) are addressed indirectly as already described in Section 6.4: 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, and if so, the planner is re-invoked with the current state of execution as the initial state (and the same goal).

7.3 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 5.4. A standard constraint solver is applied to solve the CSP to find a solution which amounts to a valid plan. We use the Choco v2.1.1 constraint programming library, which provides a large choice of implemented constraints, as well as a variety of pre-defined but also custom search methods. The solution to a CSP amounts to a partially ordered plan (i.e. one that may contain parallel actions) which is passed to the PE for execution, as already described in the previous section.

8 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 and (ii) to test the performance of the planning component with respect to the time it requires to generate the necessary IPs. 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. The service invocations are simulated, i.e. the methods kept in the service instances have a predefined behaviour, simulating different situations that we want to test.

8.1 Tests on case study

In order to test the framework we have developed on a real case-study, the WMO process shown in Figure 1 was modelled, along with the DSs shown in Figure 2. The BP specification representing the case-study is as shown in Appendix A.1,
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 1 provides an overview of the times required to generate the initial plans for all IPs shown in Figure 3, corresponding to DS1 of Figure 2, in case of a change in the applicant’s address. In all cases, the time for generating the respective initial IP is below one second. 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).

<table>
<thead>
<tr>
<th>IP</th>
<th>Plan length</th>
<th>Time for planning (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
<td>0.5</td>
</tr>
<tr>
<td>b</td>
<td>6</td>
<td>0.6</td>
</tr>
<tr>
<td>c</td>
<td>6</td>
<td>0.6</td>
</tr>
<tr>
<td>d</td>
<td>7</td>
<td>0.6</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Table 1: Performance results for generating the IPs of Figure 3

Tables 2i and 2ii present the times for computing each updated plan in the case of some possible environmental behaviour for the IPs depicted in Figures 3b and 3c, which have 2 and 3 deferred choices respectively. Re-planning is performed until the goal as specified in Section 5 is satisfied, or no solution can be found. The reported times are the average over 4 separate test runs.

The IP in Figure 3b 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 2i 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 3c covers the case where the address changes at the stage where a home modification is requested, but the request is not yet confirmed. Table 2ii 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 6.4 for a possible execution behaviour showing the exact service invocations that take place).

<table>
<thead>
<tr>
<th>State when planner is called</th>
<th>Plan length</th>
<th>Time for violation check and planning (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state</td>
<td>6</td>
<td>0.6 (optimistic plan)</td>
</tr>
<tr>
<td>&quot;Medical Advice required&quot;</td>
<td>5</td>
<td>0.3 (violation, new plan)</td>
</tr>
<tr>
<td>&quot;Rejected&quot;</td>
<td>- (no plan)</td>
<td>0.02 (violation, goal can’t be satisfied)</td>
</tr>
</tbody>
</table>

Table 2: Performance results: Re-planning times for (i) the IP of Figure 3b and (ii) the IP of Figure 3c

8.2 Scalability in 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 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.

<table>
<thead>
<tr>
<th>Planning time (in sec)</th>
<th>5 act</th>
<th>10 act</th>
<th>15 act</th>
<th>20 act</th>
<th>25 act</th>
<th>30 act</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.9</td>
<td>5.1</td>
<td>5.2</td>
<td>5.3</td>
<td>5.5</td>
<td>5.7</td>
</tr>
</tbody>
</table>

Table 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 2) 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 [17]. 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).

9 Concluding Remarks

In this paper we present an approach 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. We have studied the feasibility of our approach with respect to a real case scenario, the business process of the Dutch WMO law. We show how AI planning can be used to ensure the consistency of the process execution results in an automatic way, given a number of high-level annotations in terms of dependency scopes provided by the domain designer. The application of the approach in business domains where data can be changed by external factors, can highly benefit organizations by resolving potential inconsistencies in a way that enables a higher degree of flexibility by reducing hard-coded dependency solutions and workflow repair mechanisms.

To evaluate the feasibility of the approach, a prototype has been implemented, and tested on a real test-case from Dutch e-government. 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 inconsistencies 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.

Although the focus of the current paper 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 on which the planning framework is based on 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.

Currently, dependency scopes are specified manually by a domain expert on top of the BP specification. However, the critical sections of a BP that have to be surrounded by DSs can be identified automatically, given the structure of the BP and the service semantics concerning the input-output and the internal state variables of the service operations used by the process.

According to the approach presented herein, a recovery process is only fired in case of change events that are covered by dependency scopes. However, there may be environmental changes that compromise the consistency of the Business Process, and have not taken into consideration by the domain designer. In order to prevent potential erroneous situations resulting from such events that have not been anticipated at design time, we also consider to adopt a conservative policy that holds the execution in case a violation with respect to the BP specification is detected. This can be done by extending the orchestrator, so that it can check whether the preconditions of an action, as specified in the semantic repository, indeed hold at execution time. This will imply some extra cost for violation checks, and is similar to the approach for mismatch detection presented in [19]. These extra checks for violations during execution will also help address cases where the IP may lead to inconsistencies with respect to possible concurrent activities that were put to hold during the recovery process.

A Appendix

A.1 BP representing the WMO process

For clarity reasons, aliases are used instead of the full activity identifiers (references to service invocation methods, which reside in the SR). For instance, the decision activity name is an alias for the full identifier TenderWCSupplier.12CB.tenderDecision.

\[
\text{WHO}(\text{bpAddress}, \text{bpCid}, \text{bpEligCrit}, \text{bpMedCond}) = \\
\text{seq} \\
\text{repeat} \\
\text{seq} \\
\text{intake(itIn\_cid=bpCid)} \\
\text{guard(bpAddress)} \\
\text{homeVisit(hvIn\_address=bpAddress, hvIn\_cid=bpCid)} \\
\text{XOR} \\
\text{hvOut\_medAdvReq = TRUE ⇒ medicalAdvice(maIn\_cid:=bpCid, maIn\_homeInfo:=hvOut\_homeInfo, }
\]
mainMedCond=bpMedCond)
hvOut_medAdvReq = FALSE \Rightarrow \text{no-op}
}
decision(dcIn_cid=bpCid, dcIn_homeInfo=hvOut_homeInfo,
dcIn_medInfo=maOut_medInfo)
}
verify{
  address.county = 'Groningen': \text{achieve-maint(known(dcOut_conf))}
  address.county \neq 'Groningen': \text{terminate(achieve-maint(}
    \text{notifiedCityHall('countyChange') = TRUE))}
}
}
until(dcOut_decision \neq 'Rejected'){
XOR{
  appeal = FALSE \Rightarrow \text{terminate}
  appeal = TRUE \Rightarrow
    XOR{
      appealAffirmed = FALSE \Rightarrow \text{terminate}
      appealAffirmed = TRUE \Rightarrow \text{no-op}
    }
}
}
seq{
XOR{
  inOut_prov = 'care in kind' OR inOut_prov = 'personal budget' \Rightarrow
  XOR{
    iaOut_prov = 'personal budget' \Rightarrow \text{no-op}
    iaOut_prov = 'care in kind' \Rightarrow
      seq{
        guard(bpAddress, bpMedCond){
          sendDHRequest(sdhrIn_address=bpAddress, sdhrIn_cid=bpCid,
            sdhrIn_orderReq=hvOut_homeInfo)
          receiveDeliveryConf(dlIn_cid=bpCid, dlIn_id=orderId,
            dlIn_address=orderAddress, dlIn_delContents=orderContents)
        }
        verify{
          address.county \neq 'Groningen': \text{terminate(achieve-maint(}
            \text{notifiedCityHall('countyChange') = TRUE} \land \text{invalid(orderId))})
          address.county = 'Groningen' AND medCond \neq \text{deceased}: \text{achieve-maint(known(dlOut_conf))}
          medCond='deceased': \text{terminate(achieve-maint(}\text{invalid(orderId))})
        }
        repeat{
          seq{
            receiveInvoice(riIn_id=orderId, riIn_cid=bpCid)
            checkInvoice(ciIn_invId=riOut_invId)
          }
        }
      }
    }
}
returnInvoice(rtiIn_invId=riOut_invId)
}
}
}

itOut_prov = ‘home modification’ OR inOut_prov = ‘wheelchair’ ⇒
seq{
  guard(bpAddress, bpMedCond){
    XOR{
      iaOut_prov = ‘home modification’ ⇒
        guard(bpEligCrit){
          seq{
            repeat{
              seq{
                tenderProc(tpIn_cid=bpCid, tpIn_homeInfo=hvOut_homeInfo)
                checkTender(ctIn_cid=bpCid, ctIn_selTender=tpOut_tenderSelection)
              }
            until(ctOut_tenderOK = TRUE)
            sendOrderToSelSupplier(sosIn_cid=tpOut_tenderSel, sosIn_cid=bpCid,
                sosIn_orderInfo=tpOut_tenderContents, sosIn_address=bpAddress)
          }
        }
      }
    }
  }
  verify{
    achieve-maint(known(orderId))
  }
  iaOut_prov = ‘wheelchair’ ⇒
  seq{
    acquireReq(arIn_cid=bpCid, arIn_homeInfo=hvOut_homeInfo)
    sendOrder(soIn_cid=bpCid, soIn_orderReq=arOut_requirements,
        soIn_address=bpAddress)
  }
}
receiveDeliveryConf(dlIn_orderId=orderId)

verify{
  address.county ≠ ‘Groningen’: terminate(achieve-maint
      (notifiedCityHall(‘countyChange’) = TRUE ∧ invalid(orderId)))
  address.county = ‘Groningen’ AND medCond ≠ deceased:
      achieve-maint(known(dlOut_conf))
  medCond = ‘deceased’: terminate(achieve-maint(invalid(orderId)))
}
repeat {
  seq{
    receiveInvoice(riIn_delId=dlOut_delId, riIn_cid=cid)
    checkInvoiceWithDec(ciIn_invId=riOut_invId)
  }
  until(invOut_invoiceOK = ‘TRUE’){
    returnInvoice(rtiIn_invId=riOut_invId)
  }
}
Variable interdependencies:

\[
{\text{dependsOn}}(bpAddress) = \{hvOut\text{._homeInfo}\}
\]
\[
{\text{dependsOn}}(hvOut\text{._homeInfo}) = \{hvOut\text{._maRequired, maOut\text{._medInfo, dcOut\text{._approvalCheck, arOut\text{._requirements, tpOut\text{._tenderSelection}}}}\}
\]
\[
{\text{dependsOn}}(tpOut\text{._tenderSelection}) = \{ctOut\text{._tenderOK}\}
\]
\[
{\text{dependsOn}}(medCond) = \{maOut\text{._medInfo, dcOut\text{._approvalCheck, arOut\text{._requirements, tpOut\text{._tenderContents, ctOut\text{._tenderOK}}}}\}
\]
\[
{\text{dependsOn}}(eligCrit) = \{ctOut\text{._tenderOK}\}
\]

A.2 Planning Domain modelling the WMO process

Intake(itIn\text{._cid, itIn\text{._address}})
Prec: 
\[
itIn\text{._cid} = bpCid \land itIn\text{._address} = bpAddress
\]
Eff: 
\[
sense(itOut\text{._prov})
\]

HomeVisit(hvIn\text{._cid, hvIn\text{._address}})
Prec: 
\[
hvIn\text{._cid} = bpCid \land hvIn\text{._address} = bpAddress
\]
\[(\text{known(itOut\text{._prov})})
\]
Eff: 
\[
sense(hvOut\text{._homeInfo}) \land sense(hvOut\text{._maRequired})
\]

MedicalAdvice(maIn\text{._cid})
Prec: 
\[
maIn\text{._cid} = bpCid \land \text{known(hvOut\text{._maRequired})} \land
\]
\[
hvOut\text{._maRequired} = true \land \text{known(hvOut\text{._homeInfo})}
\]
Eff: 
\[
sense(maOut\text{._medInfo})
\]

Decision(dcIn\text{._cid, dcIn\text{._homeInfo, dcIn\text{._eligCrit, dcIn\text{._medInfo}}})
Prec: 
\[
dcIn\text{._homeInfo} = hvOut\text{._homeInfo} \land dcIn\text{._cid} = bpCid \land
\]
\[(\neg hvOut\text{._maRequired} \lor \text{known(maOut\text{._medInfo})}) \land
\]
\[(hvOut\text{._maRequired} \lor \text{true}) \land \neg \text{known(dcOut\text{._approvalCheck})} \land
\]
\[(\neg hvOut\text{._maRequired} \lor \text{dcIn\text{._medInfo} = maOut\text{._medInfo}})
\]
Eff: 
\[
sense(dcOut\text{._approvalCheck})
\]

AcquireRequirements(arIn\text{._cid, arIn\text{._homeInfo}})
Prec: 
\[
(itOut\text{._prov} = 3 \lor itOut\text{._prov} = 4) \land itOut\text{._prov} = 3 \land
\]
\[
arIn\text{._cid} = bpCid \land
\]
\[
arIn\text{._homeInfo} = hvOut\text{._homeInfo} \land
\]
\[
\text{known(dcOut\text{._approvalCheck})} \land \text{dcOut\text{._approvalCheck} = true}
\]
Eff: 
\[
sense(arOut\text{._requirements})
\]
TenderProcedure(tpIn_cid, tpIn_homeInfo)
Prec:

$$\begin{align*}
& (\text{itOut}\_\text{prov} = 3 \lor \text{itOut}\_\text{prov} = 4) \land \\
& \text{tpIn}\_\text{cid} = \text{bpCid} \land \\
& \text{tpIn}\_\text{homeInfo} = \text{hvOut}\_\text{homeInfo} \land \\
& \text{known}(\text{dcOut}\_\text{approvalCheck}) \land \text{dcOut}\_\text{approvalCheck} = \text{true}
\end{align*}$$

Eff:

$$\begin{align*}
& \text{sense}(\text{tpOut}\_\text{tenderSelection}) \land \\
& \text{assign}(\text{tpOut}\_\text{tenderContents}, \text{tpIn}\_\text{homeInfo})
\end{align*}$$

CheckTender(ctIn_cid, ctIn_selTender, ctIn_eligCrit)
Prec:

$$\begin{align*}
& \text{ctIn}\_\text{cid} = \text{bpCid} \land \\
& \text{ctIn}\_\text{selTender} = \text{tpOut}\_\text{tenderSelected}, \text{ctIn}\_\text{eligCrit} = \text{bpEligCrit}
\end{align*}$$

Eff:

$$\begin{align*}
& \text{sense}(\text{ctOut}\_\text{tenderOK}) \land \\
& (\text{ctOut}\_\text{tenderOK} = \text{false}) \Rightarrow \text{invalidate}(\text{tpOut}\_\text{tenderSelection})
\end{align*}$$

SendOrder(soIn_cid, soIn_orderInfo, soIn_address)
Prec:

$$\begin{align*}
& \text{soIn}\_\text{cid} = \text{bpCid} \land \\
& \text{soIn}\_\text{address} = \text{bpAddress} \land \\
& \text{known}(\text{arOut}\_\text{requirements}) \land \\
& \text{soIn}\_\text{orderInfo} = \text{arOut}\_\text{requirements} \land \\
& \neg\text{known}(\text{orderId})
\end{align*}$$

Eff:

$$\begin{align*}
& \text{sense}(\text{soOut}\_\text{orderId}) \land \\
& \text{assign}(\text{orderId}, \text{soOut}\_\text{orderId}) \land \\
& \text{assign}(\text{orderAddress}, \text{soIn}\_\text{address}) \land \\
& \text{assign}(\text{orderContents}, \text{soIn}\_\text{orderInfo})
\end{align*}$$

SendOrderToSelSupplier(sosIn_cid, sosIn_sid, sosIn_orderInfo, sosIn_address)
Prec:

$$\begin{align*}
& \text{sosIn}\_\text{cid} = \text{bpCid} \land \\
& \text{sosIn}\_\text{sid} = \text{tpOut}\_\text{tenderSelected} \land \\
& \text{known}(\text{ctOut}\_\text{tenderOK}) \land \\
& \text{ctOut}\_\text{tenderOK} = \text{true} \land \\
& \text{sosIn}\_\text{address} = \text{bpAddress} \land \\
& \text{sosIn}\_\text{orderInfo} = \text{tpOut}\_\text{tenderContents} \land \\
& \neg\text{known}(\text{orderId})
\end{align*}$$

Eff:

$$\begin{align*}
& \text{sense}(\text{sosOut}\_\text{orderId}) \land \\
& \text{assign}(\text{orderId}, \text{sosOut}\_\text{orderId}) \land \\
& \text{assign}(\text{orderAddress}, \text{sosIn}\_\text{address}) \land \\
& \text{assign}(\text{orderContents}, \text{sosIn}\_\text{orderInfo})
\end{align*}$$

SendDHRequest(sdhrIn_cid, sdhrIn_orderInfo, sdhrIn_address)
Prec:

$$\begin{align*}
& (\text{itOut}\_\text{prov} = 1 \lor \text{itOut}\_\text{prov} = 2) \land \\
& \text{sdhrIn}\_\text{cid} = \text{bpCid} \land \\
& \text{sdhrIn}\_\text{orderInfo} = \text{hvOut}\_\text{homeInfo} \land \\
& \text{known}(\text{dcOut}\_\text{approvalCheck}) \land \\
& \text{dcOut}\_\text{approvalCheck} = \text{true} \land \\
& \neg\text{known}(\text{orderId})
\end{align*}$$

Eff:

$$\begin{align*}
& \text{sense}(\text{sdhrOut}\_\text{orderId}) \land \\
& \text{assign}(\text{orderId}, \text{sdhrOut}\_\text{orderId}) \land \\
& \text{assign}(\text{orderAddress}, \text{sdhrIn}\_\text{address}) \land \\
& \text{assign}(\text{orderContents}, \text{sdhrIn}\_\text{orderInfo})
\end{align*}$$

DeliveryConfirmation(dlIn_cid, dlIn_id, dlIn_address, dlIn_delContents)
Prec:

$$\begin{align*}
& \text{dlIn}\_\text{cid} = \text{bpCid} \land \\
& \text{dlIn}\_\text{id} = \text{orderId} \land \\
& \text{dlIn}\_\text{address} = \text{orderAddress} \land \\
& \text{dlIn}\_\text{delContents} = \text{orderContents}
\end{align*}$$

Eff:

$$\text{sense}(\text{dlOut}\_\text{conf})$$

ReceiveInvoice(riIn_cid, riIn_id)
Prec:

$$\begin{align*}
& \text{riIn}\_\text{cid} = \text{bpCid} \land \\
& \text{riIn}\_\text{id} = \text{orderId} \land \\
& \text{known}(\text{dlOut}\_\text{conf})
\end{align*}$$

Eff:

$$\text{sense}(\text{riOut}\_\text{invId})$$

CheckInvoice(ciIn_invId)
Prec:

$$\begin{align*}
& \text{known}(\text{riOut}\_\text{invId}) \land \\
& \text{ciIn}\_\text{invId} = \text{riOut}\_\text{invId} \land \\
& \neg\text{known}(\text{ciOut}\_\text{invoiceOK})
\end{align*}$$

Eff:

$$\text{sense}(\text{ciOut}\_\text{invoiceOK})$$
ReturnInvoice(rtiIn_invId)
Prec:
known(riOut_invId) ∧ riOut_invId = rtiIn_invId ∧ ciOut_invoiceOK = false
Eff:
 invalidate(riOut_invId) ∧ invalidate(ciOut_invoiceOK)

Payment(pmIn_invId)
Prec:
(¬(itOut_prov = 1 ∨ itOut_prov = 2) ∨
(¬(itOut_prov = 1 ∨ known(dcOut_approvalCheck)) ∧ ¬(itOut_prov = 2 ∨ known(ciOut_invoiceOK)))
∧ pmIn_invId = riOut_invId)
Eff:
sense(pmOut_conf)

CancelOrder(coIn_orderId)
Prec:
known(orderId) ∧ coIn_orderId = orderId
Eff:
 invalidate(orderId)

notifyCityHall(nchIn_msg)
Prec:
∅
Eff:
sense(nchOut_sent)

References


