# Comparing Goal-Oriented and Procedural Service Orchestration*

M. Birna van Riemsdijk[1] Martin Wirsing[2]

[1] Technische Universiteit Delft, The Netherlands
m.b.vanriemsdijk@tudelft.nl
[2] Ludwig-Maximilians-Universität München, Germany
wirsing@pst.ifi.lmu.de

**Abstract.** Goals form a declarative description of the desired end result of (part of) an orchestration. A goal-oriented orchestration language is an orchestration language in which these goals are part of the language. The advantage of using goals explicitly in the language is added flexibility in handling failures. In this paper, we investigate how goal-oriented mechanisms for handling failures compare to more standard exception handling mechanisms, by providing a formally defined translation of programs in the goal-oriented orchestration language into programs in the procedural orchestration language, and proving that the procedural orchestration has the same behavior as the goal-oriented orchestration.

* This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

## 1 Introduction

Service-oriented computing is emerging as a new paradigm based on autonomous, platform-independent computational entities, called *services*, that can be described, published, and dynamically discovered and assembled. An important context in which services are used, is in service-oriented architectures (SOAs) [10]. In a SOA, services are used for facilitating the implementation of *business processes* on a business' IT infrastructure. Due to the abstraction layer introduced through the adoption of a SOA, and due to the loose coupling of services, SOA offers the potential to increase organizational agility.

This is important, since business processes and a business' IT infrastructure inherently evolve over time. Moreover, business processes are executed in dynamic, uncertain and error-prone environments [19]. It is thus important that automation environments are adaptive, both with respect to accommodating changes in business processes, as well as with respect to the execution of business processes.

In a SOA, one way to compose services to implement a business process, is by means of an *orchestration language* suitable for the execution of business processes, such as WS-BPEL[3] [18]. In order to realize the potential for agility of a SOA, such orchestration languages should facilitate adaptivity.

In this paper, we investigate the use of *goal-orientation* in orchestration languages for facilitating adaptivity. Goals form a declarative description of the desired end result of the execution of (part of) an orchestration. That is, goals describe *what* is to be achieved, as opposed to describing *how* a desired result is to be achieved. A goal-oriented orchestration language has language constructs which express the goal that is to be reached by some part of the orchestration.

Goal-oriented techniques have emerged in research on agent-oriented programming (see, e.g., [34, 16, 31, 26, 3, 17, 29]). It is generally argued that one of the advantages of the explicit use of goals in a programming language is *added flexibility* in handling failures [34, 28, Chapter 5]. The idea is essentially that goals are used to monitor the execution of statements, or *plans* in agent terminology. If the execution does not have the desired result, goals are used to *select a different plan*. This mechanism is used recursively, as plans can contain subgoals. The fact that a program and its parts contain explicit representations of the desired result of their execution thus facilitates monitoring their execution and taking appropriate measures by trying alternative courses of action if the execution fails to achieve these results.

The increased flexibility provided through the use of goal-oriented techniques is also confirmed by recent developments initiated by Whitestein Technologies AG[4], who are developing software supporting goal-oriented business process modeling and execution [6, 33, 5]. The use of goal-oriented techniques makes business process models and their implementation easier to change, it allows specification of many different plans for a particular goal expressing how to pur-

[3] BPEL stands for Business Process Execution Language.

[4] `http://www.whitestein.com/`

sue the goal in varying situations, and it allows the system to "heal itself" if problems occur, by finding alternative courses of action [33].

Moreover, as argued in [33], the use of goal-oriented techniques has significant advantages for *modeling* business processes. The use of goals fits naturally with the organization of many businesses, in which the upper management level is typically more concerned with what is to be achieved, rather than how something is to be achieved. Also, the use of goals as an abstraction increases process understandability, and it allows the specification of a wide and diverse set of solutions without the significant increase in complexity of the process definition, as found in BPEL and similar approaches. Practical experience has thus shown that goal-oriented techniques have significant advantages over more traditional approaches for business process modeling and service orchestration, such as (WS-)BPEL.

In this paper, we investigate the use of goal-oriented techniques in an orchestration language from a *theoretical* perspective. We investigate how the goal-oriented orchestration language of [32] can be translated into a program in a more traditional procedural orchestration language that has *provably* the same behavior. Since increased flexibility in handling failures is one of the main advantages of the use of goal-oriented techniques, we focus on a comparison of the failure handling mechanism of the goal-oriented orchestration language with the exception handling mechanism of the procedural orchestration language. The exception handling mechanism of the latter is inspired by that of WS-BPEL.

It will become clear that the programming patterns resulting from the translation of the goal-oriented orchestration language into the procedural orchestration language do not increase understandability of the code. Since expressing the kind of abstractions used in the goal-oriented orchestration language in a procedural orchestration language thus leads to complex orchestration definitions, and since the use of goal-oriented techniques has significant practical advantages, we argue that goal-oriented abstractions are worth considering as language constructs of an orchestration language.

The organization of this paper is as follows. In Section 2, we introduce the running example that we use to illustrate the definitions of the orchestration languages. In Sections 3 and 4, we define the goal-oriented and procedural orchestration languages, respectively. We present the translation of the goal-oriented orchestration language into the procedural orchestration language, and the result stating the correctness of the translation, in Section 5. In Section 6, we discuss related work on planning, and we conclude the paper in Section 7.

## 2 Running Example: Engineering Change Request

The running example we use in this paper is from the domain of management of engineering processes. Engineering processes, such as designing a car, take many years and are typically very complex with many alternative execution paths. One of the most crucial sub-processes is Engineering Change Request (ECR) [25]. The ECR process covers the processing of an ECR from the initial proposal for a change, through its evaluation regarding costs, technical feasibility, compliance to laws and regulations, etc., to its approval or rejection.

We illustrate our definitions by specifying a simplified version of parts of the ECR process in our orchestration languages. Whitestein has used the ECR process of Daimler AG[5], which broadly follows the standard ECR process specified by SASIG[6] [25], as a case study for their goal-oriented business process management software [6, 33, 5]. Since our goal-oriented orchestration language is closely related to the goal-oriented business process modeling language of Whitestein, our specification of the ECR process resembles theirs.

It is important to note that the specification of the ECR process in our goal-oriented orchestration language is necessarily simplified, since the language is not meant to be a full-fledged orchestration language. Its purpose is to investigate the semantic foundations[7] of goal-oriented orchestration languages. To keep the language simple, the specification of goals and services is based on propositional logic. In [30], we propose a formal specification framework for services that is based on description logic, and we refer to [8] for the description of a goal-oriented agent programming language and platform based on first-order logic, which uses similar goal-oriented techniques as the ones we use in this paper.

The main purpose of the ECR process is to have a change request managed, i.e., the ECR should be specified and it should be decided whether to approve or reject it. This top-level goal can be subdivided into four subgoals: the ECR has to be *initiated*, it has to be *created* and described in detail, it has to be *analyzed* and evaluated, and a *decision* has to be made as to whether the requested change will be implemented. The full process as modeled by Whitestein contains 60 goals [5].

In this paper, we investigate the goal-oriented orchestration language of [32] (with some small modifications). Its main language construct is the so-called *plan selection rule*. Using a plan selection rule, one can specify which plan may be executed for achieving a certain goal in a particular context. These rules thus consist of the goal that is to be achieved, a plan that specifies how to reach the goal, and a condition specifying in which context the plan can be executed, and have the following form: $goal \mid contextCondition \Rightarrow plan$. Plans can contain subgoals to express that these subgoals must be reached in order to achieve the goal of the plan selection rule.

**Example 1** *(ECR specified and decided)* In this example, we show how a plan selection rule can be used for specifying how the top-level goal of the ECR process (*ECRspec&decided*) can be reached. Goals are preceded by an exclamation mark, and $\gg$ is used to program sequential composition. The context condition specifies that the plan can be executed if a proposal for an ECR has been received, or if

[5] `http://www.daimler.com/`

[6] Strategic Automotive Product Data Standards Industry Group

[7] "Semantic" is here meant in the sense of "semantics of programming languages" [9], not in the sense of "semantic web technology" [2].

an ECR needs to be revised.

$$\begin{array}{l}\texttt{!ECRspec\&decided} \mid \texttt{ECRproposal} \vee \texttt{ECRrevision} \Rightarrow \\ \qquad\qquad \texttt{!ECRinitiated} \gg \texttt{!ECRcreated} \gg \texttt{!ECRanalyzed} \gg \texttt{!ECRdecided}\end{array}$$

This plan selection rule specifies, that in order to reach the goal *ECRspec&decided*, the four subgoals *ECRinitiated*, *ECRcreated*, *ECRanalyzed*, and *ECRdecided* have to be achieved in sequence. The rule does not specify how these subgoals are to be achieved. This is in turn done using plan selection rules, of which we will give some examples in the next section. △

## 3 Goal-Oriented Orchestration Language

In this section, we present the goal-oriented orchestration language of [32], with some small modifications. We present its syntax in Section 3.1, and in Section 3.2 we present the informal semantics and the part of the formal semantics that is relevant for failure handling. We refer to [32] and Appendix A for more details and explanation. While the language presented in this section is mostly the language from [32], in this paper we illustrate the language using many examples from the ECR domain, and we focus the presentation and discussion on failure handling.

### 3.1 Syntax

A program in the goal-oriented orchestration language is called an agent. The main components of an agent are, loosely speaking, a representation of the context in which it operates, a set of top-level goals, and a set of plan selection rules. The context, which in agent terminology is called a *belief base*, is a consistent set of propositional formulas, denoted by $\sigma$. The main purpose of the belief base is to store information about the context, but it can also be used to store information internal to the agent. The set of top-level goals is called the *goal base*, and is denoted by $\gamma$. A goal is denoted by $\kappa$ and can be either an achievement goal $!p$ (where $p$ is an atom)[8], representing that the agent wants to achieve a situation in which $p$ holds, or a test goal $?p$, representing that the agent wants to know whether $p$ holds. Formally, the belief base and goal base are defined as follows.

**Definition 1** *(belief base and goal base)* Assume a standard language of propositional logic $\mathcal{L}$, defined over a set of propositional atoms $\mathsf{Atom}$. The set of belief bases $\Sigma$ with typical element $\sigma$ is defined as $\{\sigma \mid \sigma \subseteq \mathcal{L},\ \sigma \not\models \bot\}$. The set of goals $\mathcal{L}_\mathsf{G}$ with typical element $\kappa$ is defined as $\{?p, !p \mid p \in \mathsf{Atom}\}$. A goal base $\gamma$ is a subset of $\mathcal{L}_\mathsf{G}$, i.e., $\gamma \subseteq \mathcal{L}_\mathsf{G}$.

[8] In [32], we used arbitrary propositional formulas for the representation of goals, but for reasons of simplicity we use atoms here.

Plan selection rules are formally denoted as $\kappa \mid \beta \Rightarrow \pi$, where $\kappa$ is the goal of the plan selection rule, $\beta$ is a propositional formula representing a condition on the beliefs (context) that should hold for the rule to be applicable, and $\pi$ is a plan.

The basic elements of plans are *internal actions*, typically denoted by $a$, which can be used for making changes to the belief base, *subgoals*, and *service calls*. A service call has the form $sn^r(act_\phi, act_\kappa)$, where $sn$ is the name of the service that is to be called (which is $d$ if a service is to be dynamically discovered), $act_\kappa$ represents the goal that is to be achieved through calling the service, and $act_\phi$ is (or should be instantiated with) a propositional formula representing input to the service. If only the goal is needed for calling the service, the input parameter is omitted. The revision parameter $r$ can be $np$ (non-persistent), meaning that the output of the service call is not stored in the belief base, or $p$ (persistent), meaning that the output is stored in the belief base.

If a service is called, the output returned by the service can be used in the remainder of a plan using the sequential composition operator $>x>$ , where the output is bound to the variable $x$. In this way, the output of one service can conveniently be used as the input for another service. If the output is not used, or if no result is expected, then $\gg$ can be used for sequential composition. This sequential composition operator is inspired by a similar construct in the orchestration language Orc [7]. The execution of an internal action does not yield an output, i.e., internal actions are always composed using $\gg$. Subgoals *can* yield output. In this case, the output loosely speaking consists of those parts of the belief base that express the achievement of the subgoal.

For example, assume the belief base contains the formula `ECRinitialDoc` $\rightarrow$ `ECRinitiated` (denoted as $\phi_1$), representing that if a document exists describing an initial ECR, then the ECR has been initiated, and the formula `ECRinitialDoc` (denoted as $\phi_2$), representing that a document describing an initial ECR exists.[9] Then, the subgoal `!ECRinitiated` is believed to be achieved since `ECRinitiated` follows from the belief base, and the conjunction of $\phi_1$ and $\phi_2$ would be returned as the output of this subgoal.

The formal definition of the syntax of plans is given below, where $x$ is a variable name, $\kappa$ is as in Definition 1, and $a$ is an internal action. For simplifying the definition, we omit the distinction between $>x>$ and $\gg$.

$$\begin{array}{ll} act_\phi ::= x \mid \phi & \quad b ::= a \mid \kappa \mid sn^r(act_\phi, act_\kappa) \\ act_\kappa ::= x \mid \kappa & \quad \pi ::= b \mid b >x> \pi \end{array}$$

In Example 1, we showed a plan selection rule specifying how the top-level goal of the ECR process can be reached. The plan of the rule contained several subgoals, for which in turn plan selection rules have to be specified. In the following example, we show how the plan selection rule for the subgoal of initiation of an ECR can be specified.

[9] Note that due to the fact that we use propositional logic, we cannot express conveniently that the document for a certain ECR concerns that ECR. For practical use, the language will have to be extended to a first-order variant, allowing the use of variables as parameters of goals.

**Example 2** *(initiation of an ECR)* The following plan selection rule specifies that a plan for initiation of an ECR can be executed, if a proposal for an ECR exists. The plan specifies that a service for creating an initial ECR is called, and the output of this service, i.e., an initial ECR (*init*), is passed to a service that decides whether this ECR can be pursued.

$$\begin{aligned}&\texttt{!ECRinitiated} \mid \texttt{ECRproposal} \Rightarrow \\ &\qquad \texttt{createInit}(\texttt{ECRproposal}, \texttt{!initECRcreated})^{np} > \mathit{init} > \\ &\qquad\qquad \texttt{decidePursuit}(\mathit{init}, \texttt{!ECRinitiated})^{p}\end{aligned}$$

If it is decided to pursue the ECR, the output of the latter service is meant to be a document describing an initial ECR (`ECRinitialDoc`). The revision parameter $p$ of the service `decidePursuit` specifies that the output of this service is stored in the belief base. In combination with a belief base containing the formula $\phi_1$ as described above, this would mean that the subgoal `!ECRinitiated` is then achieved. △

## 3.2 Semantics

In this section, we present the semantics of our goal-oriented orchestration language. First, we present the basic mechanisms involved in "normal" execution, i.e., in case no failures occur. Then, we present the formal semantics of failure handling mechanisms.

**Normal Execution** The main execution mechanism of an agent in our goal-oriented orchestration language is the application of plan selection rules to goals in the goal base or subgoals in plans. The application of plan selection rules is formalized using the notion of a stack. Each element of the stack represents, broadly speaking, the application of a plan selection rule to a particular (sub)goal. The initial stack element is created by applying a plan selection rule to a top-level goal in the goal base, and other stack elements are created every time a subgoal is encountered in the plan of the top element of a stack.

A stack element has the form $(\pi, \kappa, \mathsf{PS})$, where $\kappa$ is the (sub)goal to which the plan selection rule has been applied, $\pi$ is the plan currently being executed in order to achieve $\kappa$, and $\mathsf{PS}$ is the set of plan selection rules that have not yet been tried in order to achieve $\kappa$. That is, if a plan selection rule from $\mathsf{PS}$ has been applied to try to reach $\kappa$, it is removed from $\mathsf{PS}$. This is a simple heuristic to make sure the agent does not keep trying to reach a goal over and over again with the same plan selection rules. More advanced mechanisms could be used, but investigating those are beyond the scope of this paper.

**Example 3** *(stack)* Assume an ECR agent with a set of plan selection rules $\mathsf{PS}$ that includes the rules of Examples 1 and 2, to which we will refer as $\rho_1$ and $\rho_2$, respectively. Furthermore, assume the agent has the goal `!ECRspec&decided` as top-level goal in its goal base, and the formula `ECRproposal` in its belief base.

The rule $\rho_1$ can then be applied to the goal !`ECRspec&decided`, resulting in the following initial stack element (where $\pi_1$ is the plan of $\rho_1$).

$$(\pi_1, !\texttt{ECRspec\&decided}, \mathsf{PS} \setminus \{\rho_1\}) \tag{1}$$

Note that $\rho_1$ is removed from the set of plan selection rules that can still be used for trying to achieve the goal !`ECRspec&decided`, since $\rho_1$ has just been applied to try to reach that goal.

The first element of $\pi_1$ is the subgoal !`ECRinitiated`, meaning that in order to achieve !`ECRspec&decided`, this subgoal has to be achieved first. Since `ECRproposal` is in the belief base, the rule $\rho_2$ can then be applied to that subgoal. This results in the creation of another stack element on top of the initial stack element, yielding the following stack (where $\pi_2$ is the plan of $\rho_2$).

$$(\pi_2, !\texttt{ECRinitiated}, \mathsf{PS} \setminus \{\rho_2\}).(\pi_1, !\texttt{ECRspec\&decided}, \mathsf{PS} \setminus \{\rho_1\}) \tag{2}$$

The set of plan selection rules of the top element of the stack is formed by the set of plan selection rules of the agent, minus the plan selection rule $\rho_2$ that has been applied to create the stack element. $\triangle$

In order to define the semantics formally, we need to introduce several notions. An agent is formally defined as a tuple $\langle \sigma_0, \gamma_0, \mathsf{PS}_{\mathcal{A}}, \mathcal{T} \rangle$, where $\sigma_0$ is the (initial) belief base, $\gamma_0$ is the initial goal base, $\mathsf{PS}_{\mathcal{A}}$ is a set of plan selection rules, and $\mathcal{T}$ is a partial *belief update function* $(\mathsf{InternalAction} \times \Sigma) \to \Sigma$ (where $\mathsf{InternalAction}$ is the set of internal actions of the agent and $\Sigma$ is a set of belief bases) which specifies how the belief base changes, if an internal action is executed by the agent. This function is introduced as usual [28] for technical convenience. A configuration of an agent has the form $\langle \sigma, \gamma, \mathrm{St}, \mathsf{PS}_{\mathcal{A}}, \mathcal{T} \rangle$, where St is the stack. The initial configuration of an agent $\langle \sigma_0, \gamma_0, \mathsf{PS}_{\mathcal{A}}, \mathcal{T} \rangle$ is $\langle \sigma_0, \gamma_0, E, \mathsf{PS}_{\mathcal{A}}, \mathcal{T} \rangle$, where $E$ denotes an empty stack.

The formal semantics of our goal-oriented orchestration language is defined using a transition system [23]. A transition system for a programming language consists of a set of axioms and transition rules for deriving transitions for this language. A transition is a transformation of one configuration into another and it corresponds to a single computation step. The transition rules specify how to execute the top element of a stack, and we leave out $\mathsf{PS}_{\mathcal{A}}$ and $\mathcal{T}$ from configurations for reasons of presentation (and these do not change during computation).

In the initial configuration of an agent, the stack containing the plans that are being executed is empty, since no plan selection rules have been applied yet. In order to initialize the stack, a plan selection rule is applied to a goal in the goal base as follows, where $\mathsf{PS}_{\mathcal{A}}$ are the plan selection rules of the agent that is executing. We use a predicate applicable$(\rho, \kappa, \sigma)$ to denote that plan selection rule $\rho$ is applicable to goal $\kappa$, given belief base $\sigma$ (see Appendix A for the definition).

**Definition 2** *(initialization of stack)*

$$\frac{\kappa' \mid \beta \Rightarrow \pi \in \mathsf{PS}_{\mathcal{A}} \quad \kappa \in \gamma \quad \text{applicable}(\kappa' \mid \beta \Rightarrow \pi, \kappa, \sigma) \quad \mathsf{PS}' = \mathsf{PS}_{\mathcal{A}} \setminus \{\kappa' \mid \beta \Rightarrow \pi\}}{\langle \sigma, \gamma, E \rangle \to \langle \sigma, \gamma, (\pi, \kappa, \mathsf{PS}') \rangle}$$

Stack elements are thus created through the application of plan selection rules. A stack element is popped just after a service call or the execution of an internal action if the goal of the stack element is reached, or if the goal is unreachable, meaning that there are no applicable plan selection rules. An example of the former case is presented next, and the latter case will be explained in more detail in the sequel.

**Example 4** *(popping a stack element: goal reached)* Consider stack (2). In this situation, the plan of the top element of the stack, i.e., $\pi_2$, will be executed. In a normal execution, the goal of this stack element, i.e., `!ECRinitiated`, will be reached after the execution of $\pi_2$. In that case, the top element of the stack will be popped, and the subgoal `!ECRinitiated` is removed from $\pi_1$, since this has been achieved. Let $\pi_1' = \texttt{!ECRcreated} \gg \texttt{!ECRanalyzed} \gg \texttt{!ECRdecided}$ be the remaining part of $\pi_1$. We then have the following stack.

$$(\pi_1', \texttt{!ECRspec\&decided}, \mathsf{PS} \setminus \{\rho_1\}) \tag{3}$$

Now, the first subgoal of $\pi_1'$ should be achieved, i.e., `!ECRcreated`, for which a plan selection rule of $\mathsf{PS} \setminus \{\rho_1\}$ should be applied. △

Top-level goals from the goal base are removed as soon as they are believed to be achieved, i.e., as soon as they follow logically from the belief base.

**Failure Handling** In the goal-oriented orchestration language, a *failure* is not only caused by abnormalities in trying to execute some operation, as in more traditional languages, but also by *being unsuccessful in reaching a goal.* In particular, if a service is called and returns some output, the call is only considered to be successful if the goal of the service call is reached through the output that is returned. That is, even if the service returns a "normal" or non-exceptional result, the service call can still be regarded as having failed. Such situations are not unlikely to occur, especially if services are automatically discovered at runtime. It might, e.g., be the case that the service description was not accurate, resulting in an unsatisfactory result. These kinds of failures are typically neither considered nor dealt with in more classical programming paradigms, in which a failure or exception is normally caused by the fact that some operation could not be executed properly.

Our goal-oriented orchestration language *handles failures of service calls* by repeatedly trying to find matching services for a service call (in particular if services are to be discovered) until the goal of the service call is reached, or there are no more matching services.[10] If the latter happens, the service call has failed definitively, in which case the plan containing the service call is considered to have failed and the plan is dropped.

[10] One might argue that a comprehensive failure handling mechanism should include compensation, but this is without the scope of this paper. We refer to [4] for a formal approach to compensations.

**Example 5** *(failure of service calls)* We consider the subgoal !`ECRanalyzed` of the plan for reaching the top-level goal !`ECRspec&dec` (Example 1). In order to reach this subgoal, the costs of the ECR have to be assessed. One way to assess the costs, is by estimating the costs. In general, there are several ways of estimating costs, and there are correspondingly several services that can do this estimation. Alternatively, services may be provided by people, and there may be several people who are capable of making estimations (see [1] for an extension of WS-BPEL to include the provision of services by people, called BPEL4People). In order to find a service that can do cost estimations, a discovery mechanism can be used.

The top element of the stack, when the orchestration has come to the point where the costs of an ECR have to be estimated, may look as follows, where $\mathsf{PS}$ is some set of plan selection rules.

$$(d(!\texttt{ECRcostsEstimated}), !\texttt{costsAssessed}, \mathsf{PS}) \tag{4}$$

The $d$ in the service call $d(!\texttt{ECRcostsEstimated})$ represents that a service should be *discovered*, in this case for doing an estimation of costs of an ECR. We assume that a set of services with service descriptions exists in some repository. In Appendix A, we provide a simple definition of when a service description matches a service call. We refer to [30] for a more expressive framework for service description using description logic, and for a corresponding definition of matching.

When a service call such as $d(!\texttt{ECRcostsEstimated})$ occurs in a plan, a matching service is discovered. If a matching service can be discovered, the service is called, and its output is returned. The output is compared against the goal of the service call. If the goal has been reached, the agent continues the execution of the rest of the plan. Otherwise, it tries to find another service that matches the service call. It may be the case that no (more) services can be found that match the service call. In that case, the plan is dropped, yielding the following stack element, where $\epsilon$ represents an empty plan.

$$(\epsilon, !\texttt{costsAssessed}, \mathsf{PS}) \tag{5}$$

$\triangle$

A formal definition of dropping a plan when no more matching services exist for a service call, is specified in Definition 3 below. The service call construct $sn^r(\phi, \kappa')$ (we assume variables are instantiated when the service is called) is annotated with a set of service descriptions $S$ which represents those services from the repository that have not yet been called, and the result $x_0$ of the last service call.[11]

In this setting, services are assumed to return a propositional formula. The predicate $\text{ach}(\kappa, \sigma, x_o)$ holds iff the goal $\kappa$ is achieved with respect to belief base

[11] The definition of the syntax of plans can easily be extended to allow for these annotations, but for simplicity and brevity, we do not do this here.

$\sigma$ and the service call result $x_0$. In case $\kappa$ is an achievement goal, it is achieved if the goal follows from the belief base after it is updated with $x_0$. In case $\kappa$ is a test goal, it is achieved if the goal or its negation follow from $x_0$. The idea is that the belief base should not be taken into account when evaluating the achievement of a test goal, as a service is called in order to check whether some piece of information is accurate. Then it does not matter whether the agent already believes something about this information. The predicate $\text{match}(sn(\phi,\kappa),\sigma,sd)$ holds iff the service with service description $sd$ matches with the service call $sn(\phi,\kappa)$, given the belief base $\sigma$. The transition rules specify how to execute the top of a stack. We refer to Definition 19 for a transition rule specifying that a complete stack can be executed by executing its top.

**Definition 3** *(plan failure)*

$$\frac{\neg\text{ach}(\kappa',\sigma,x_o) \quad \neg\exists sd \in S : \text{match}(sn(\phi,\kappa'),\sigma,sd)}{\langle \sigma,\gamma,(sn^r(\phi,\kappa')[S,x_o] >x> \pi,\kappa,\mathsf{PS})\rangle \rightarrow \langle \sigma,\gamma,(\epsilon,\kappa,\mathsf{PS})\rangle}$$

We have defined a similar rule for the case that an internal action cannot be executed, in which case the plan is dropped as well (see Appendix A.2). Plans are thus dropped if something goes wrong, i.e., an empty plan indicates a plan failure.[12]

While the handling of failures of service calls is done by trying to call other matching services, the *handling of plan failures* is done by using plan selection rules to select *alternative* plans for reaching a (sub)goal.

**Example 6** *(handling plan failure)* We consider the goal `!costsAssessed` of having the costs of an ECR assessed (see also Example 5). In general, there are two ways of assessing the costs: calculating the costs, and estimating the costs. Correspondingly, we specify two alternative plan selection rules for assessing costs. The first rule can be applied if no estimation-based approval of an ECR is carried out. The plan specifies that a service is to be discovered for calculating the costs. The second rule can be applied if estimation-based approval is carried out. The plan specifies that a service is to be discovered for estimating the costs.

$$\begin{array}{lll} \texttt{!costsAssessed} \mid \neg\texttt{estBasedApproval} & \Rightarrow d(\texttt{!ECRcostsCalculated}) \\ \texttt{!costsAssessed} \mid \texttt{estBasedApproval} & \Rightarrow d(\texttt{!ECRcostsEstimated}) \end{array} \quad (6)$$

These plan selection rules are mutually exclusive, given the context condition of the rules. Deciding whether an estimation-based approval of an ECR is carried out, is usually done in the part of the orchestration preceding the subgoal of assessing the costs. However, the decision of carrying out an estimation-based

[12] It can also be the case that a plan is *completely executed* resulting in an empty plan, i.e., without the plan having been dropped. This also indicates failure, namely failure in reaching the goal of the stack element. The stack element would have been popped immediately if its goal would have been reached after an action execution or service call, not giving rise to a stack element with an empty plan.

approval or not, may be reconsidered after an attempt at calculating or estimating the costs has failed. This is specified using the following plan selection rule.

$$\begin{aligned}&\texttt{!costsAssessed} \mid \texttt{true} \Rightarrow \\ &\qquad \texttt{reconsiderApprovalStrategy}(\texttt{!approvalStrategyReconsidered})\end{aligned} \tag{7}$$

Note that this would require an ordering among rules, since rule (7) should be applied only after one of the rules of (6) has been applied, and has failed to reach the goal `!costsAssessed`. If a previous decision to do an estimation-based approval or not is reconsidered after execution of (7), the *alternative* rule of (6) that has not yet been applied, can be tried. △

The application of an alternative plan selection rule to achieve a particular (sub)goal after a plan has failed, is formally specified by the transition rule below. Note that the plan selection rule that is applied is removed from the set of available plan selection rules $\mathsf{PS}$, which prevents the agent from trying the same rule over and over again. Moreover, note that the fact that we store the subgoal that the agent is trying to reach in the stack elements, facilitates the selection of alternative plans to reach this goal. If we would not have such a representation, it would be more difficult to determine what to do if something went wrong.

**Definition 4** *(apply rule after plan failure)* Below, $\mathsf{PS}' = \mathsf{PS} \setminus \{\kappa \mid \beta \Rightarrow \pi\}$.

$$\frac{\kappa \mid \beta \Rightarrow \pi \in \mathsf{PS} \quad \neg\mathrm{ach}(\kappa, \sigma, \top) \quad \sigma \models \beta}{\langle \sigma, \gamma, (\epsilon, \kappa, \mathsf{PS})\rangle \rightarrow \langle \sigma, \gamma, (\pi, \kappa, \mathsf{PS}')\rangle}$$

At some point, it may be the case that all plan selection rules for a particular subgoal have been tried, without the subgoal being reached. In this case, the subgoal is considered to have failed definitively. In that case, the top element of the stack containing the failed subgoal is popped, and the plan of the new top element of the stack is dropped.

**Example 7** *(subgoal failure)* We consider again stack (2) of Example 3, which is the stack after a plan selection rule for achieving the top-level goal `!ECRspec&decided` has been applied, followed by the application of a plan selection rule for achieving the subgoal `!ECRinitiated`.

$$(\pi_2, \texttt{!ECRinitiated}, \mathsf{PS} \setminus \{\rho_2\}).(\pi_1, \texttt{!ECRspec\&decided}, \mathsf{PS} \setminus \{\rho_1\}) \tag{8}$$

Now assume that $\pi_2$ is executed, but it is decided not to pursue the proposal for the ECR. In this case, the subgoal `!ECRinitiated` is not reached after the execution of $\pi_2$. Typically, there will be no alternative plans for achieving that subgoal, which means that it has failed. Then, the top element of the stack is popped, and the plan $\pi_1$ of the new top element is dropped, yielding the following stack.

$$(\epsilon, \texttt{!ECRspec\&decided}, \mathsf{PS} \setminus \{\rho_1\}) \tag{9}$$

The plan $\pi_1$ is dropped, because it contained the subgoal !`ECRinitiated`, and the agent has failed to achieve that subgoal. This means that $\pi_1$ has failed as well. If an alternative plan selection rule for !`ECRspec&decided` exists in $\mathsf{PS} \setminus \{\rho_1\}$, that rule can consecutively be applied in this situation. If this is not the case, the agent has failed to achieve the top-level goal !`ECRspec&decided`. $\triangle$

This is specified formally below. Consider a (top-part of a) stack $(\epsilon, \kappa, \mathsf{PS}).(\kappa >x> \pi, \kappa', \mathsf{PS}')$, in which the plan of its top element is empty, and assume there are *no* plan selection rules applicable to the subgoal $\kappa$ of this stack element. In this case, the top element is popped, and the plan $\kappa >x> \pi$ that contains $\kappa$ is dropped from the new top element. Consecutively, the agent can try another plan for reaching the subgoal $\kappa'$, or, if there are no applicable plan selection rules, the stack element with subgoal $\kappa'$ is popped as well, etc.

**Definition 5** *(subgoal failure)*

$$\frac{\neg\exists \rho \in \mathsf{PS} : \text{applicable}(\rho, \kappa, \sigma)}{\langle \sigma, \gamma, (\epsilon, \kappa, \mathsf{PS}).(\kappa >x> \pi, \kappa', \mathsf{PS}') \rangle \to \langle \sigma, \gamma, (\epsilon, \kappa', \mathsf{PS}') \rangle}$$

## 4 Procedural Orchestration Language

The main ingredients of our procedural orchestration language are standard features of procedural languages, i.e., assignment, test, procedure call, and an exception handling mechanism. The particular instantiations of these features are tailored towards the translation of the goal-oriented orchestration language in the procedural orchestration language. Further, the language includes a construct for service calls, similar to the corresponding one in the goal-oriented orchestration language.

The state of configurations of the language consists of a belief base and goal base as also used in the goal-oriented orchestration language. However, the goal base is interpreted simply as a set of data elements, i.e., it is a normal data structure that does not have the semantics of its counterpart in the goal-oriented orchestration language. The syntax of statements in the procedural language is formally defined below, where $e$ is an exception name, $x$ is a variable name, and $act_\phi$, $act_\kappa$ are as in Section 3.1.

$$\begin{array}{lll}
\kappa & ::= & ?p \mid !p \\
v & ::= & \mathit{true} \mid \mathit{false} \mid \phi \mid \kappa \\
t & ::= & \phi? \mid (x = v)? \mid \mathit{ach}(act_\kappa, x)? \mid \mathit{not}\ t \mid t \wedge t' \\
act & ::= & x \mid v \\
exp & ::= & v \mid \kappa(act_1, \ldots, act_n) \mid sn^r(act_\phi, act_\kappa) \mid \mathit{base}(act_\kappa) \\
b & ::= & a \mid x := exp \mid t \mid \mathsf{return}\ act \mid \mathsf{throw}\ e \\
\pi & ::= & b \mid b; \pi \mid \pi + \pi' \mid \mathtt{while}\ t\ \mathtt{do}\ \pi\ \mathtt{od}
\end{array}$$

The language of procedure names $\kappa$ is the same as the language of goals of the goal-oriented orchestration language. Procedures may use local variables, typically denoted by $x$. These local variables may have a value $v$, which is *true*, *false*, a formula $\phi$, or a procedure name $\kappa$. Tests $t$ can be global tests on the belief base $\phi$? (note the difference with test goals $?p$, which can only be fulfilled through service calls), local tests $(x = v)$? which can be used for testing the value of a variable, or $ach(act_\kappa, x)$, which tests whether the goal $act_\kappa$ is achieved with respect to the value of the variable $x$.

Expressions *exp* are values, procedure calls $\kappa(act_1, \ldots, act_n)$, service calls, or a call to a predefined function $base(act_\kappa)$, which returns a conjunction of formulas from the belief base from which $act_\kappa$ follows, or *false* if $act_\kappa$ does not follow. Intuitively, this represents how $act_\kappa$ is achieved. An elementary statement $b$ can be an action $a$ to change the belief base (as in the goal-oriented orchestration language), an assignment $x := exp$ to change the value of local variable $x$, a test, the returning of a variable, and the throwing of an exception. Composed statements are formed by sequential composition, non-deterministic choice, or a `while` construct.

The exception handling mechanism that we use is inspired by the exception handling mechanism in the service orchestration language WS-BPEL [18]. In WS-BPEL, exception handlers are associated with a scope of a business process. If a fault occurs in a scope and the scope contains a matching handler, the process specified by the handler is executed.[13] If there is no handler, the exception is passed to the enclosing scope. In the context of our procedural language, the scope is formed by procedures, i.e., each procedure call gives rise to a new scope. Therefore, we associate exception handlers to procedures, as defined below. A handler contains the name of the exception that it handles, and a statement that should be executed if the relevant exception is thrown.

**Definition 6** *(procedures and exception handlers)* A procedure has the form $\kappa(x_1, \ldots, x_n) \Rightarrow \pi$. Exception handlers, typically denoted by $h$, have the form $e.\mathsf{Handler} \Rightarrow \pi$, where $e$ is an exception name. A procedure definition is a procedure accompanied with a possibly empty set of exception handlers, denoted by $[\kappa \Rightarrow \pi, H]$, where $H$ is a set of exception handlers.

The semantics is defined by means of a transition system. We use stacks to define the mechanism of calling procedures, analogously to the way this was done for applying plan selection rules. Each stack element $(\pi, \theta, H)$ corresponds to a procedure call, where $\pi$ is the statement that still needs to be executed, $\theta$ is a substitution specifying which values have been assigned to which local variables, and $H$ is the set of exception handlers of the procedure that was called and for which the stack element was created. The set of handlers of a stack element does not change during computation.

A configuration $\langle \sigma, \gamma, \mathrm{St}, \mathsf{P}, \mathcal{T} \rangle$ consists of a belief base $\sigma$ and goal base $\gamma$ (together forming the global state), a stack St, a set of procedure definitions $\mathsf{P}$, and a belief update function $\mathcal{T}$. A program $\langle \sigma_0, \gamma_0, \pi_0, \mathsf{P}, \mathcal{T} \rangle$ has the initial configuration $\langle \sigma_0, \gamma_0, (\pi_0, \emptyset, \emptyset), \mathsf{P}, \mathcal{T} \rangle$. Analogously to the goal-oriented orchestration language, we omit the procedure definitions and the belief update function from configurations in the transition rules below.

[13] Additionally, WS-BPEL has a compensation mechanism (see also [20]), which is, however, outside the scope of this paper.

We only show the transition rules for exception handling. The semantics of the other constructs is as one would expect, and for formal details we refer to Appendix B. The semantics of procedure calls is a simple call-by-value semantics. The first transition rule below expresses that if an exception $e$ is thrown from within a stack element, and the stack element contains a handler $e.\mathsf{Handler} \Rightarrow \pi'$ for this exception, then the statement $\pi'$ is executed instead of the statement from which the exception was thrown. If the stack element does not contain a handler for $e$, the exception is passed to the stack element one level lower in the stack.

**Definition 7** *(throwing exceptions)*

$$\frac{e.\mathsf{Handler} \Rightarrow \pi' \in H}{\langle \sigma, \gamma, (\mathsf{throw}\ e; \pi, \theta, H)\rangle \rightsquigarrow \langle \sigma, \gamma, (\pi', \theta, H)\rangle}$$

$$\frac{\neg\exists h' \in H' : h' \text{ is of the form } e.\mathsf{Handler} \Rightarrow \pi''}{\langle \sigma, \gamma, (\mathsf{throw}\ e; \pi', \theta', H').(\pi, \theta, H)\rangle \rightsquigarrow \langle \sigma, \gamma, (\mathsf{throw}\ e, \theta, H)\rangle}$$

## 5 Translation and Correctness Result

In this section, we show how the goal-oriented orchestration language can be translated to a procedural orchestration. This translation shows, first of all, *how* goal-oriented orchestration, and in particular its failure handling mechanism, is related to a more standard procedural orchestration language and its exception handling mechanism. Moreover, it shows that the programming patterns resulting from the translation do not increase understandability of the code. As stated in [12] in a more general context, the problem with programming patterns is that "they are an obstacle to an understanding of programs for both human readers and programming-processing programs".[14]

**Example 8** *(translation of the ECR program)* In this example, we show how the part of the goal-oriented ECR orchestration related to the subgoal of assessing the costs of an ECR (see Example 5) could be programmed in the procedural orchestration language.

We combine the plan selection rules for the goal `!costsAssessed` as presented in Example 5 into one procedure using non-deterministic choice between the translated rules. The idea is then to translate any occurrence of the subgoal `!costsAssessed` in a plan, in this case in the plan for `!ECRanalyzed`, into a corresponding procedure call.

The service calls occurring in the plans of the plan selection rules are translated into a while-loop, that tries calling matching services until the goal of the service call is reached, or until no more matching services are available. If no matching service is available, an exception `!costsAssessed`.planFailedExc is

[14] The term "programming patterns" should not be confused with "design patterns". While the former are computational in nature, the latter are concerned with software architecture.

thrown, as the plan has failed and should be aborted in this case. The corresponding exception handler calls the procedure `costsAssessed` recursively, so that *another* plan can be tried to achieve the goal `costsAssessed`.

We use the variables $tried_1, tried_2$, and $tried_3$ to record which of the plans (as specified in the three plan selection rules) have already been tried to reach the goal. If all plans have been tried and/or none are applicable, the exception `ECRanalyzed`.planFailedExc is thrown which is caught lower down in the procedure call stack in the procedure `ECRanalyzed`. This is done since the procedure `costsAssessed` was called from the procedure `ECRanalyzed`, and a failure to achieve the goal of assessing costs should lead to failure of the plan that is being executed to achieve the goal of getting the ECR analyzed.

We omit some aspects from the example program, such as removing services that have been tried but failed from the set of tried services, for reasons of simplicity.

```
costsAssessed($tried_1, tried_2, tried_3$) $\Rightarrow$
  (($tried_1 = false$)? $\wedge$ $\neg$estBasedApproval;
  ($tried_1 := true$); $x := base$(!ECRcostsCalculated);
  while $not\ ach$(!ECRcostsCalculated, $x$) do
    $x := d$(!ECRcostsCalculated);
    (($x = nomatch$)?; throw costsAssessed.planFailedExc) +
    ($not$($x = nomatch$))? od;
  return $x$)
+
  (($tried_2 = false$)? $\wedge$ estBasedApproval;
  ($tried_2 := true$); $x := base$(!ECRcostsEstimated);
  while $not\ ach$(!ECRcostsCalculated, $x$) do
    $x := d$(!ECRcostsCalculated);
    (($x = nomatch$)?; throw costsAssessed.planFailedExc) +
    ($not$($x = nomatch$))? od;
  return $x$)
+
  (($tried_3 = false$)?;
  ($tried_3 := true$); $x := base$(!approvalStrategyReconsidered);
  while $not\ ach$(!approvalStrategyReconsidered, $x$) do
    $x :=$ reconsiderApprovalStrategy(!approvalStrategyReconsidered);
    (($x = nomatch$)?; throw costsAssessed.planFailedExc) +
    ($not$($x = nomatch$))? od;
  return $x$)
+
($not$(($tried_1 = false$)? $\wedge$ $\neg$estBasedApproval) $\wedge$
$not$(($tried_2 = false$)? $\wedge$ estBasedApproval) $\wedge$
$not$(($tried_3 = false$)?);
throw ECRanalyzed.planFailedExc)

costsAssessed.planFailedExc.Handler $\Rightarrow$ costsAssessed($tried_1, tried_2, tried_3$)
```

$\triangle$

Since expressing the kind of abstractions used in the goal-oriented orchestration language in a procedural orchestration language thus leads to complex orchestration definitions, and since the use of goal-oriented techniques has significant practical advantages as discussed in Section 1, we argue that the kind of abstractions as used in the goal-oriented orchestration language are worth considering as language constructs of an orchestration language.

As our procedural orchestration language and WS-BPEL are comparable in the sense that they have a similar exception handling mechanism, and both are imperative languages without goal-oriented constructs, we conjecture that an implementation of goal-oriented orchestration patterns in WS-BPEL will be similarly involved as in our procedural orchestration language.

The rest of the section will be concerned with the definition of a translation of arbitrary goal-oriented orchestrations to procedural orchestrations, and proving the correctness of this translation. We present the most important parts of the translation, i.e., the translation of plan selection rules (Section 5.1) and the translation of plans (Section 5.2). For the full technical details of the translation, we refer to Appendix C. The theorem expressing the correctness of the translation is presented in Section 5.3.

## 5.1 Translation of Plan Selection Rules

Example 8 already hints at how a translation of a goal-oriented orchestration into a procedural one might be defined. That is, all plan selection rules for a certain goal are translated into one procedure that has this goal as the procedure name. The body of the procedure resulting from the translation of a set of plan selection rules, broadly speaking, consists of a non-deterministic choice between the translated plans of the relevant plan selection rules, guarded by tests on the belief base corresponding with the guards of the plan selection rules.[15] The translation of plans is specified through the function $u_\kappa$ (see Section 5.2).

**Definition 8** *(translating plan selection rules)* Without loss of generality, assume that variables in the goal-oriented orchestration language are not the reserved variables $tried_i$. Let $\mathsf{PS}$ be a set of plan selection rules. Let $\mathsf{PS}_\kappa$ be defined as $\{\kappa \mid \beta \Rightarrow \pi \ : \ \kappa \mid \beta \Rightarrow \pi \in \mathsf{PS}\}$ and let $n = |\mathsf{PS}_\kappa|$. We assume an ordering on the elements of $\mathsf{PS}_\kappa$ as follows: $\{\kappa \mid \beta_1 \Rightarrow \pi_1, \ldots, \kappa \mid \beta_n \Rightarrow \pi_n\}$. The translation function $t(\mathsf{PS}_\kappa)$ for translating $\mathsf{PS}_\kappa$ into one procedure definition is defined as follows, where we use the notation $+_{1 \leq i \leq n} expr_i$ to denote $expr_1 + \ldots + expr_n$.

$$
\begin{array}{l}
[\kappa(tried_1, \ldots, tried_n, from) \Rightarrow \\
\quad this := \kappa; \\
\quad (+_{1 \leq i \leq n}((tried_i = false)? \wedge \beta_i?; tried_i := true; u_\kappa(\pi_i); [\alpha_{fail}]\ \mathsf{throw}\ \kappa.\mathsf{planFailedExc}) + \\
\quad (\bigwedge_{1 \leq i \leq n} not((tried_i = false)? \wedge \beta_i?); [\alpha_{fail}]\ \mathsf{throw}\ from.\mathsf{planFailedExc})), \\
\{\kappa.\mathsf{planFailedExc.Handler} \Rightarrow x_f := \kappa(tried_1, \ldots, tried_n, from); \mathsf{return}\ x_f\}]
\end{array}
$$

[15] In the example we used if-then-else constructs rather than non-deterministic choice, but in order to make the translation correct, we need non-deterministic choice to match the non-determinism of the goal-oriented orchestration language in selecting plan selection rules.

It needs to be recorded which plans have already been tried to reach a certain goal, as done in the goal-oriented orchestration language through storing the set of not yet tried plan selection rules $\mathsf{PS}$ in each stack element. This is done using the variables $\mathit{tried}_i$, which can be *true* or *false*, depending on whether the statement corresponding with the plan of the $\mathrm{i}^{\mathrm{th}}$ plan selection rule has already been tried or not, respectively.

Each situation of failure of the goal-oriented orchestration language as analyzed in detail in Section 3.2, corresponds to the throwing of an exception in the procedural language. That is, we throw a planFailedExc if a plan has been executed completely, as this means that the goal to be achieved by this plan was not reached. Further, a planFailedExc is thrown if all plans have been tried and/or none are applicable (as the belief condition does not hold), corresponding to subgoal failure (Definition 5). The throwing of an exception in case a service call fails is specified in Definition 9.

We annotate each planFailedExc with the name of the procedure in which the exception should be handled. The exception should be handled either in the procedure $\kappa$ from which it was thrown (in case another plan should be selected for achieving the goal of the procedure), or in the procedure from which $\kappa$ was first called (as passed to $\kappa$ through the variable *from*). The latter case represents the failure of a subgoal, and it corresponds to the popping of a stack element in the goal-oriented orchestration language (Definition 5).

We associate with each procedure $\kappa$ a handler for the exception $\kappa$.planFailedExc. This handler specifies that the procedure should be called recursively with the variables $\mathit{tried}_i$ as parameters. This recursive call makes sure that if a plan fails, another plan is tried which has not been tried yet (Definition 4). The annotations $[\alpha_{fail}]$ are "program points" which mark particular points in the code. They are used for defining and proving the correctness of the translation, and do not affect the semantics.

Note that the programmer thus needs to program the throwing of exceptions and their handlers explicitly in the procedural orchestration language, while the identification of situations of failure and the consecutive course of action is part of the semantics of the goal-oriented orchestration language.

## 5.2 Translation of Plans

The next definition specifies the function $u_\kappa$, which translates plans of the bodies of plan selection rules with head $\kappa$ into statements of the procedural language. The function is also used to translate the plan of a stack element with subgoal $\kappa$. The annotations (program points) of the form $[\alpha]$ are introduced similarly to the way this was done in Definition 8 for defining and proving the correctness of the translation.

**Definition 9** *(translating plans to statements)* We define a function $u_\kappa(\pi)$ where $\kappa$ is the head of the plan selection rule of which the body $\pi$ is translated, or the goal of the stack element containing $\pi$. Let $\mathsf{PS}_{\kappa'} = \{\kappa' \mid \beta' \Rightarrow \pi' : \kappa' \mid \beta' \Rightarrow \pi' \in \mathsf{PS}\}$, let $n' = \mid \mathsf{PS}_{\kappa'} \mid$, let $\mathit{false}_{1,\ldots,n'}$ be a vector of length $n'$

of parameters being the value *false*, and let $S_{\mathcal{O}}$ be the set of available service descriptions, and let $sd_{sn}$ be the service description of the service called for service call $sn^r(act_\phi, act_{\kappa'})$.

$$
\begin{aligned}
u_\kappa(\kappa' \mathbin{>x>} \pi) &= [\alpha_{\kappa'}]\ ((ach(\kappa')?; x := base(\kappa')) + \\
&\quad (not\ ach(\kappa')?; x := \kappa'(false_{1,\dots,n'}, \kappa))); u_\kappa(\pi) \\
u_\kappa(a \gg \pi) &= [\alpha_a]\ a; ((ach(\kappa)?; x := base(\kappa); \mathsf{return}\ x) + \\
&\quad (not\ ach(\kappa)?; u_\kappa(\pi))) \\
u_\kappa(sn^r(act_\phi, act_{\kappa'}) \mathbin{>x>} \pi) &= [\alpha_{sn1}]\ x := base(act_{\kappa'}); ((ach(act_{\kappa'}, x); u_\kappa(\pi)) + \\
&\quad (not\ ach(act_{\kappa'}, x)?; S := S_{\mathcal{O}}; \\
&\quad \mathtt{while}\ not\ ach(act_{\kappa'}, x)\ \mathtt{do}\ [\alpha_{sn2}]\ x := sn^r(act_\phi, act_{\kappa'}); \\
&\quad ((x = nomatch)?; [\alpha_{fail}]\ \mathsf{throw}\ \kappa.\mathsf{planFailedExc}) + \\
&\quad (not(x = nomatch)?; S := S \setminus \{sd_{sn}\})\ \mathtt{od}); \\
&\quad [\alpha_{sn3}]\ ((ach(\kappa, x)?; \mathsf{return}\ x) + \\
&\quad\qquad (not\ ach(\kappa, x)?; u_\kappa(\pi)))
\end{aligned}
$$

A subgoal $\kappa' \mathbin{>x>} \pi$ is translated into a non-deterministic choice, followed by the translation of $\pi$. The non-deterministic choice expresses that if the goal $\kappa'$ is already reached before calling the procedure $\kappa'$, $x$ gets a value through the function $base(\kappa')$. If $\kappa'$ is not yet achieved, the procedure $\kappa'$ is called, which returns a value (a propositional formula) that expresses how $\kappa'$ was achieved or an exception in case $\kappa'$ could not be achieved. The actual parameters for the procedure $\kappa'$ are a series of *false* values, expressing that no plans have yet been tried to reach $\kappa'$, and the last parameter is the subgoal $\kappa$, which is the goal to be reached through execution of the statement $u_\kappa(\kappa' \mathbin{>x>} \pi)$ (as we are translating plan selection rules with head $\kappa$).

The translation of an action $a$ expresses that $a$ should be executed, and, depending on whether the goal $\kappa$ is reached, the orchestration returns or continues with the execution of $u_\kappa(\pi)$. The translation of a service call $sn^r(act_\phi, act_{\kappa'})$ defines that matching services are called until $act_{\kappa'}$ is reached, or there are no more matching services. If the latter is the case, a planFailedExc is thrown (corresponding to Definition 3).

Using the translation functions as defined above, we have defined a function $v$ (see Appendix C for its definition) for translating agents of the goal-oriented orchestration language into procedural programs in the procedural orchestration language. This function $v$ uses the function $t$ of Definition 8 to translate plan selection rules to procedures. Moreover, an initialization procedure is added, which is called from the initial statement of the resulting procedural program. The purpose of the initialization procedure is to initiate the pursuit of goals of the goal base (corresponding to Definition 2). The procedure is defined such that the program terminates if the goal base is empty, in correspondence with the semantics of the goal-oriented orchestration.

### 5.3 Correctness of Translation

We show, broadly speaking, that an agent in the goal-oriented orchestration language has the same behavior as its translation in the procedural orchestration language. We do this by showing that each run of an agent $\mathcal{A}$ has a matching run of agent $v(\mathcal{A})$ and vice versa. A run of $\mathcal{A}$ matches a run of $v(\mathcal{A})$, loosely speaking, if each configuration of the former has a matching configuration in the latter (in the right order). Each transition in a run of $\mathcal{A}$ is matched by a series of transitions in a run of $v(\mathcal{A})$, i.e., not each configuration of a run of $v(\mathcal{A})$ has a matching configuration in the corresponding run of $\mathcal{A}$.

The definition of when a procedural configuration matches a goal-oriented configuration is provided by a function $z$ (see Appendix C for its definition), which translates a configuration of the procedural orchestration language into a configuration of the goal-oriented language. The function cannot be defined the other way around, as procedural configurations contain certain implementation details that do not have a counterpart in goal-oriented configurations.

The function $z$ translates in particular procedural stacks into goal-oriented stacks by translating statements of stack elements to plans. The function uses the substitution of stack elements to determine the goal of the resulting goal-oriented stack element (recorded in the variable *this*), and to determine which plan selection rules have not yet been tried to reach the goal (recorded in the variables $\mathit{tried}_i$).

The function does not translate arbitrary statements to plans, but only those that have reached a program point of the form $[\alpha]$. These statements correspond to a plan in the goal-oriented orchestration language. The idea is that each transition in a run of $\mathcal{A}$ is matched by a series of transitions in a run of $v(\mathcal{A})$, where this series of transitions starts in a configuration where the statement is at a program point, and ends in a configuration where the statement has reached the next program point.

The correctness of the translation is formulated formally below. We refer to Appendix C for the complete proof.

**Theorem 1** Let $\mathcal{A}$ be a program in the goal-oriented orchestration language with initial configuration $c_0$ and $v(\mathcal{A})$ the translation of $\mathcal{A}$. Then it holds for any run $c_0 \to c_1 \to \ldots$ that there exist indices $0 = p_0 < p_1 < \ldots$ and configurations $d_0, d_1, \ldots$ such that $d_0 \rightsquigarrow d_1 \rightsquigarrow \ldots$ is a run in the procedural orchestration language, $d_0$ is the initial configuration of $v(\mathcal{A})$, and for all $p_i$ with $i \geq 0$ it holds that $z(d_{p_i}) = c_i$.

Let $P$ be a program in the procedural orchestration language with initial configuration $d_0$ such that there is some program $\mathcal{A}$ of the goal-oriented orchestration language with $v(\mathcal{A}) = P$. Then it holds for any run $d_0 \rightsquigarrow d_1 \rightsquigarrow \ldots$ that there exist indices $0 = p_0 < p_1 < \ldots$ and configurations $c_0, c_1, \ldots$, such that $c_0 \to c_1 \to \ldots$ is a run in the goal-oriented orchestration language, $c_0$ is the initial configuration of $\mathcal{A}$, and for all $p_i$ with $i \geq 0$, it holds that $z(d_{p_i}) = c_i$.

*Sketch of proof:* The main tool for proving the correctness result, is the insertion of program points in statements of the procedural program resulting from the translation of a program in the goal-oriented orchestration language. We specify that a configuration $d$ of a procedural program is at program point $\alpha$, if the statement of the top element of the stack of $d$ is of the form $[\alpha]\pi$, denoted by $\alpha : d$. We then define the relation $\Rightarrow$ as follows. If $\alpha_0 : d_0 \rightsquigarrow \ldots \rightsquigarrow \alpha_n : d_n$ such that no configuration $d_i$ with $0 < i < n$ is at a program point, then $\alpha_0 : d_0 \Rightarrow \alpha_n : d_n$. The idea is that we can partition a run from the procedural program into parts that go from one program point to the next, where such a part is abstractly denoted by $\alpha_0 : d_0 \Rightarrow \alpha_n : d_n$. The indices $0 = p_0 < p_1 < \ldots$ referred to in the theorem then correspond to those configurations that are at a program point.

The first part of the theorem is proven by induction over the length of runs in the goal-oriented orchestration language, and by structural induction (reasoning by cases) to prove the result for arbitrary configurations. That is, we show that if $z(\alpha_i : d_{p_i}) = c_i$ for $i \geq 0$, there is a $p_{i+1}$ such that $\alpha_i : d_{p_i} \Rightarrow \alpha_{i+1} : d_{p_{i+1}}$ and $z(\alpha_{i+1} : d_{p_{i+1}}) = c_{i+1}$, where $c_i \rightarrow c_{i+1}$. The induction basis is provided by showing that $z(\alpha_{init} : d_0) = c_0$.

In order to prove the second part of the theorem for a program $P$ such that $v(\mathcal{A}) = P$ for some $\mathcal{A}$, we take $0 = p_0 < p_1 < \ldots$ to be indices such that for each $d_{p_i}$ with $i \geq 0$, $d_{p_i}$ is at a program point and $\alpha_i : d_{p_i} \Rightarrow \alpha_{i+1} : d_{p_{i+1}}$ holds. We then prove that for all $p_i$ with $i \geq 0$, it holds that $z(d_{p_i}) = c_i$ and $c_0 \rightarrow c_1 \rightarrow \ldots$ is a run of $\mathcal{A}$, by induction over the length of $d_{p_0} \rightsquigarrow^* d_{p_1} \rightsquigarrow^* \ldots$. We have that $p_0 = 0$ and $z(d_{p_0}) = c_0$, providing the induction basis. For $i \geq 0$ and $z(d_{p_i}) = c_i$, we show for $c_{i+1}$ with $z(d_{p_{i+1}}) = c_{i+1}$ that $c_i \rightarrow c_{i+1}$. $\square$

## 6 Related Work: Planning

In this section, we highlight a related area of research that has not been addressed so far in this paper, i.e., planning. *Planning* can be used for service composition as an alternative to *programming* a service composition in an orchestration language. The classical AI planning problem is to search for a plan (a sequence of actions) to get from the current state to a goal state, given a set of action specifications [13, 14]. It has been observed that automated composition of services can be viewed as a planning problem (see, e.g., [21, 22]), where services are viewed as actions (with appropriate specifications), and composition requirements form planning goals.

The main difference between planning and programming approaches is that planning involves *search* for a plan that reaches a particular goal, while plans are specified by the programmer in the other case. Search is typically performed off-line, before execution of the plan. In order to reduce the search space, planning approaches often extend the classical planning problem by making use of pre-specified plan templates (see, e.g., [11, 15]). Such planning approaches bear a strong resemblance to programming approaches. In fact, in [24], HTN (Hierarchical Task Network) style planning [11] is incorporated into an agent programming language by building on the underlying similarities between the programming language and HTN planning.

Despite these commonalities between planning and programming approaches, the difference remains that planning involves off-line search for a plan that reaches a particular goal, while plans specified as programs are executed directly on-line, without first checking whether a plan will reach a goal. This difference leads to different research questions being addressed in each case. In particular, research on planning is often concerned with the development of efficient algorithms for performing the search, while research on programming approaches is concerned with the specification of appropriate execution semantics of programs. In this paper, we have been concerned with orchestration languages, taking a programming approach to service composition. Consequently, our focus has been on the investigation of the (operational) semantics of these languages.

## 7 Conclusion

In this paper, we have shown how the goal-oriented orchestration language of [32] can be correctly translated to a procedural orchestration language. As we have argued that the failure handling mechanism of the goal-oriented orchestration language is one of its main advantages, it is important to investigate whether a similar mechanism cannot be implemented just as easily in a more traditional language. As we have shown, however, the translation is non-trivial and the programming patterns resulting from the translation do not increase understandability of the code. We thus argue that the kind of abstractions as used in the goal-oriented orchestration language are worth considering as language constructs of an orchestration language.

An important topic for future research is the extension of the goal-oriented orchestration language towards more practically usable versions, e.g., by making use of description logic instead of propositional logic (see [30] for a formal service specification and matchmaking framework based on description logic). This will allow us to experiment with the language in order to further investigate the usefulness of such a language in the domain of service orchestration. The usefulness of goal-oriented abstractions will not only have to be investigated on the level of orchestration languages, but also on the modeling level. The approach for goal-oriented business process modeling of [6, 33, 5] (see Section 1) suggests that goal-oriented techniques can be used for business process modeling. In future work, we aim to investigate this in more detail. Moreover, we want to investigate whether the KAOS goal-oriented requirements engineering methodology [27] can be adapted to fit the goal-oriented orchestration language.

## References

1. Active Endpoints Inc., Adobe Systems Inc., BEA Systems Inc., International Business Machines Corporation, Oracle Inc., and SAP AG. WS-BPEL Extension for People (BPEL4People), Version 1.0, 2007. `https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/30c6f5b5-ef02-2a10-c8b5-cc1147f4d58c`.
2. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

3. L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal representation for BDI agent systems. In *Programming multiagent systems, second international workshop (ProMAS'04)*, volume 3346 of *LNAI*, pages 44–65. Springer, Berlin, 2005.
4. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 209–220, 2005.
5. B. Burmeister, M. Arnold, F. Copaciu, and G. Rimassa. BDI-agents for agile goal-oriented business processes. In *Proceedings of the seventh international joint conference on autonomous agents and multiagent systems (AAMAS'08): Industry and Applications Track*, pages 37–44, Estoril, 2008.
6. B. Burmeister, H.-P. Steiert, T. Bauer, and H. Baumgärtel. Agile processes through goal- and context-oriented business process modeling. In *Business Process Management Workshops*, volume 4103 of *LNCS*, pages 217–228. Springer, 2006.
7. W. R. Cook and J. Misra. Computation orchestration: A basis for wide-area computing, 2007. To appear in the Journal on Software and System Modeling.
8. M. Dastani, M. B. van Riemsdijk, and J.-J. Ch. Meyer. Programming multi-agent systems in 3APL. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.
9. J. de Bakker. *Mathematical Theory of Program Correctness*. Series in Computer Science. Prentice-Hall International, London, 1980.
10. T. Erl. *Service-Oriented Architecture: Concepts, Technologies, and Design*. Prentice Hall, 2005.
11. K. Erol, J. Hendler, and D. S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, volume 2, pages 1123–1128, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.
12. M. Felleisen. On the expressive power of programming languages. In N. Jones, editor, *ESOP '90 3rd European Symposium on Programming, Copenhagen, Denmark*, volume 432, pages 134–151. Springer-Verlag, New York, N.Y., 1990.
13. R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
14. M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.
15. G. d. Giacomo, Y. Lespérance, and H. Levesque. *ConGolog*, a Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.
16. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming with declarative goals. In *Intelligent Agents VI - Proceedings of the 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL'2000)*, Lecture Notes in AI. Springer, Berlin, 2001.
17. J. F. Hübner, R. H. Bordini, and M. Wooldridge. Declarative goal patterns for AgentSpeak. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, 2006.
18. M. Juric, P. Sarang, and B. Mathew. *Business Process Execution Language for Web Services 2nd Edition*. Packt Publishing, 2006.
19. M. Klein, C. Dellarocas, and A. Bernstein. Journal of computer supported cooperative work: Special issue on adaptive workflow systems. 9(3-4), 2000.
20. R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL, 2006. To appear in Journal of Logic and Algebraic Programming (JLAP), Elsevier press.

21. S. McIlraith and T. Son. Adapting golog for composition of semantic web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR'02)*, pages 482–493, 2002.

22. M. Pistore, P. Traverso, and P. Bertoli. Automated composition of web services by planning in asynchronous domains. In *Proceedings of the fifth international conference on automated planning and scheduling (ICAPS'05)*, pages 2–11, 2005.

23. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

24. S. Sardina, L. P. de Silva, and L. Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS'06)*, pages 1001–1008, Hakodate, Japan, 2006. ACM Press.

25. SASIG: strategic automotive product data standards industry group. Ecm recommendation, part 1 (ecr), 2008. `http://www.prostep.org/de/downloads/empfehlungen-standards.html`.

26. J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, 2003.

27. A. van Lamsweerde and E. Letier. From object orientation to goal orientation: a paradigm shift for requirements engineering. In *Radical Innovations of Software and Systems Engineering in the Future: 9th International Workshop (RISSEF'02)*, volume 2941 of *LNCS*, pages 325–340, London, UK, 2004. Springer-Verlag.

28. M. B. van Riemsdijk. *Cognitive Agent Programming: A Semantic Approach*. PhD thesis, 2006.

29. M. B. van Riemsdijk, M. Dastani, J.-J. Ch. Meyer, and F. S. de Boer. Goal-oriented modularity in agent programming. In *Proceedings of the fifth international joint conference on autonomous agents and multiagent systems (AAMAS'06)*, pages 1271–1278, Hakodate, 2006.

30. M. B. van Riemsdijk, R. Hennicker, and M. Wirsing. Service specification and matchmaking using description logic: An approach based on institutions. In *12th International Conference on Algebraic Methodology and Software Technology (AMAST'08)*, volume 5140 of *LNCS*, pages 392–406. Springer-Verlag, 2008.

31. M. B. van Riemsdijk, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in Dribble: from beliefs to goals using plans. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 393–400, Melbourne, 2003.

32. M. B. van Riemsdijk and M. Wirsing. Using goals for flexible service orchestration: A first step. In J. Huang, R. Kowalczyk, Z. Maamar, D. Martin, I. Mueller, S. Stoutenburg, and K. Sycara, editors, *Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE'07)*, volume 4504 of *LNCS*, pages 31–48, 2007.

33. Whitestein Technologies AG. Goal-oriented autonomic business process management: Whitepaper, 2007. `http://www.whitestein.com`.

34. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *Proceedings of the eighth international conference on principles of knowledge respresentation and reasoning (KR2002)*, Toulouse, 2002.

## A Goal-Oriented Orchestration Language

### A.1 Service Description

#### Syntax

**Definition 10** *(service description)* Throughout this paper we assume a language of propositional logic $\mathcal{L}$ with typical element $\phi$ that is based on a set of atoms $\mathsf{Atom}$, where $\top, \bot \in \mathsf{Atom}$ and $\mathit{failure} \notin \mathsf{Atom}$. Moreover, we define a language $\mathcal{L}_o = \mathcal{L} \cup \{\mathit{failure}\}$ for describing the output of services. We use $\phi$ not only to denote elements of $\mathcal{L}$, but also of $\mathcal{L}_o$, but if the latter is meant, this will be indicated explicitly. Let $N_{sn}$ with typical element $sn$ be a set of service names such that $d \notin N_{sn}$.

The set of *service descriptions* $\mathcal{S}$ with typical element $sd$ is then defined as follows:

$$\{\langle sn, \mathsf{in}, \mathsf{out}, \mathsf{prec}, \mathsf{eff}\rangle \mid \mathsf{in} \subseteq \mathsf{Atom}, \mathsf{out} \subseteq \mathcal{L}_o, \mathit{failure} \in \mathsf{out}, \mathsf{prec} \in \mathcal{L} \text{ and } \mathsf{eff} \subseteq \mathcal{L}\}.$$

**Definition 11** *(information providing and world altering services)* Let $\langle sn, \mathsf{in}, \mathsf{out}, \mathsf{prec}, \mathsf{eff}\rangle$ be a service description. This service description is an information providing service iff $\mathsf{eff} \equiv \top$ and for each $\phi \neq \mathit{failure} \in \mathsf{out}$, there is a $\phi' \in \mathsf{out}$ such that $\phi' \equiv \neg\phi$. The service description is a world altering service iff $\mathsf{out} \setminus \{\mathit{failure}\} \equiv \mathsf{eff}$, i.e., if $\phi \in \mathsf{out} \setminus \{\mathit{failure}\}$, then $\exists \phi' \in \mathsf{eff} : \phi' \equiv \phi$, and vice versa.

#### Semantics

**Definition 12** *(matching a service to a goal)* Assume a function $\mathit{atoms} : \mathcal{L} \to \wp(\mathsf{Atom})$ that takes a formula from $\mathcal{L}$ and yields the set of atoms that occur in the formula. Let $sd = \langle sn', \mathsf{in}, \mathsf{out}, \mathsf{prec}, \mathsf{eff}\rangle$ be a service description. Then the matching predicate $\mathrm{match}(sn(\phi, \kappa), \sigma, sd)$, which takes a service call $sn(\phi, \kappa)$, a belief base $\sigma$, and a service description $sd$, is defined as follows if $sn \neq d$.

$$\begin{array}{lcl} \mathrm{match}(sn(\phi, ?\phi'), \sigma, sd) & \Leftrightarrow & sd \text{ is information providing and } sn = sn' \text{ and} \\ & & \mathrm{atoms}(\phi), \{p\} \subseteq \mathsf{in} \text{ and } \sigma \not\models \neg\mathsf{prec} \text{ and} \\ & & \exists \mathsf{out}' \subseteq \mathsf{out} : \mathsf{out}' \not\models \bot \text{ and } \mathsf{out}' \models \phi' \\ \mathrm{match}(sn(\phi, !p), \sigma, sd) & \Leftrightarrow & sd \text{ is world altering and } sn = sn' \text{ and} \\ & & \mathrm{atoms}(\phi), \{p\} \subseteq \mathsf{in} \text{ and } \sigma \not\models \neg\mathsf{prec} \text{ and} \\ & & \exists \mathsf{eff}' \subseteq \mathsf{eff} : \mathsf{eff}' \not\models \bot \text{ and } \mathsf{eff}' \models \phi' \end{array}$$

If $sn = d$, then the same definition applies, but the requirement that $sn = sn'$ is dropped.

**Definition 13** *(semantics of service execution)* Let $sd = \langle sn, \mathsf{in}, \mathsf{out}, \mathsf{prec}, \mathsf{eff}\rangle$ be a service description. The predicate $\mathit{ret}$ is then defined as follows.

$$\begin{array}{l} \mathrm{ret}(sd, \phi) \Leftrightarrow \phi \equiv \mathit{failure} \text{ or} \\ \qquad \exists \mathsf{out}' \subseteq \mathsf{out} \setminus \{\mathit{failure}\} : (\mathsf{out}' \not\models \bot \text{ and } \bigwedge_{\phi_o \in \mathsf{out}'} \phi_o \equiv \phi) \end{array}$$

## A.2 Orchestration Language

### Syntax

**Definition 14** *(belief base and goal base)* The set of belief bases $\Sigma$ with typical element $\sigma$ is defined as $\{\sigma \mid \sigma \subseteq \mathcal{L},\ \sigma \not\models \bot\}$. The set of goals $\mathcal{L}_{\mathsf{G}}$ with typical element $\kappa$ is defined as $\{?p, !p \mid p \in \mathsf{Atom}\}$. A goal base $\gamma$ is a subset of $\mathcal{L}_{\mathsf{G}}$, i.e., $\gamma \subseteq \mathcal{L}_{\mathsf{G}}$.

**Definition 15** *(plan)* Let $a$ be an internal action, let $x$ be a variable, and let $N^+_{sn}$ be defined as $N_{sn} \cup \{d\}$.[16] Let $sn \in N^+_{sn}$, $r \in \{np, p\}$, $\phi \in \mathcal{L}$ and $\kappa \in \mathcal{L}_{\mathsf{G}}$. Then the set of plans $\mathsf{Plan}$ with typical element $\pi$ is defined as follows, where $b$ stands for basic plan element.

$$\begin{array}{ll} act_\phi ::= x \mid \phi & b ::= a \mid \kappa \mid sn^r(act_\phi, act_\kappa) \\ act_\kappa ::= x \mid \kappa & \pi ::= b \mid b >x> \pi \end{array}$$

**Definition 16** *(plan selection rules)* The set of plan selection rules $\mathcal{R}_{\mathsf{PS}}$ is defined as $\{\kappa \mid \beta \Rightarrow \pi \ :\ \kappa \in \mathcal{L}_{\mathsf{G}},\ \beta \in \mathcal{L},\ \pi \in \mathsf{Plan}\}$[17].

**Definition 17** *(stack)* The set of stacks $\mathsf{Stack}$ with typical element $St$ to denote arbitrary stacks, and $st$ to denote single elements of a stack, is defined as follows, where $\pi \in \mathsf{Plan}$, $\kappa \in \mathcal{L}_{\mathsf{G}}$, and $\mathsf{PS} \subseteq \mathcal{R}_{\mathsf{PS}}$.

$$\begin{array}{l} \mathrm{st} ::= (\pi, \kappa, \mathsf{PS}) \\ \mathrm{St} ::= \mathrm{st} \mid \mathrm{st}.\mathrm{St} \end{array}$$

$E$ is used to denote the empty stack (or the empty stack element), and $E.\mathrm{St}$ is identified with St.

**Definition 18** *(agent)* An agent $\mathcal{A}$ is a tuple $\langle \sigma_0, \gamma_0, \mathsf{PS}, \mathcal{T} \rangle$ where $\sigma_0 \in \Sigma$ is the belief base, $\gamma_0 \subseteq \mathcal{L}_{\mathsf{G}}$ is the goal base, $\mathsf{PS} \subseteq \mathcal{R}_{\mathsf{PS}}$ is a finite set of plan selection rules, and $\mathcal{T}$ is a partial function of type $(\mathsf{InternalAction} \times \Sigma) \to \Sigma$ and specifies the belief update resulting from the execution of internal actions. The initial configuration of this agent is $\langle \sigma_0, \gamma_0, E, \mathsf{PS}, \mathcal{T} \rangle$, i.e., it has initially an empty stack.

### Semantics

**Definition 19** *(stack execution)* Let $st \neq E$.

$$\frac{\langle \sigma, \gamma, \mathrm{st} \rangle \to \langle \sigma', \gamma', \mathrm{st}' \rangle}{\langle \sigma, \gamma, \mathrm{st}.\mathrm{St} \rangle \to \langle \sigma', \gamma', \mathrm{st}'.\mathrm{St} \rangle}$$

[16] We use $sn$ as typical element of $N_{sn}$ and of $N^+_{sn}$. It will generally be clear from the context which is meant, and otherwise it will be indicated explicitly.

[17] We use the notation $\{\ldots\ :\ \ldots\}$ instead of $\{\ldots \mid \ldots\}$ to define sets, to prevent confusing usage of the symbol $\mid$ in this definition.

**Definition 20** *(belief revision function)* In the following, we assume a partial belief revision function $brev : (\wp(\mathcal{L}) \times \wp(\mathcal{L})) \rightarrow (\mathcal{L} \rightarrow \wp(\mathcal{L}))$. The function $brev(\sigma, x)$ should satisfy the following constraints on behavior: $brev(\sigma, x) = \sigma'$ where $\sigma' \models x$ and $\sigma' \not\models \bot$; if $\sigma \models x$, then $brev(\sigma, x) = \sigma$.

**Definition 21** *(semantics of goal achievement)* The semantics of goal achievement is defined as a predicate $\text{ach}(\kappa, \sigma, x)$ that takes a goal $\kappa$, a belief base $\sigma$, and a propositional formula $x \in \mathcal{L}$ that represents the result against which $\kappa$ should be checked.

$$\begin{aligned}&\text{ach}(?p, \sigma, x) \Leftrightarrow brev(\sigma, x) = \sigma' \text{ and } (x \models p \text{ or } x \models \neg p) \text{ and } x \neq \mathit{failure}\\ &\text{ach}(!p, \sigma, x) \;\Leftrightarrow brev(\sigma, x) = \sigma' \text{ and } \sigma' \models p \text{ and } x \neq \mathit{failure}\end{aligned}$$

**Definition 22** *(applicability of plan selection rule)* We define a predicate $\text{applicable}(\rho, \kappa, \sigma)$ that takes a plan selection rule $\rho$, a goal $\kappa$, and a belief base $\sigma$ as follows, where "$\cdot$" stands for ? or !.

$$\text{applicable}(\cdot\, p \mid \beta \Rightarrow \pi, \cdot\, \phi, \sigma) \Leftrightarrow \phi \models p \text{ and } \sigma \models \beta \text{ and } \neg\text{ach}(\cdot\, p, \sigma, \top)$$

In the following, $S_{\mathcal{A}}$ is the set of service descriptions we assume to be available to the agent, and $\mathsf{PS}_{\mathcal{A}}$ is its set of plan selection rules.

**Definition 23** *(initialization of stack)*

$$\frac{\kappa' \mid \beta \Rightarrow \pi \in \mathsf{PS}_{\mathcal{A}} \quad \kappa \in \gamma \quad \text{applicable}(\kappa' \mid \beta \Rightarrow \pi, \kappa, \sigma) \quad \mathsf{PS}' = \mathsf{PS}_{\mathcal{A}} \setminus \{\kappa' \mid \beta \Rightarrow \pi\}}{\langle \sigma, \gamma, E \rangle \rightarrow \langle \sigma, \gamma, (\pi, \kappa, \mathsf{PS}') \rangle}$$

**Definition 24** *(calling services)*

$$\frac{\neg\text{ach}(!\phi', \sigma, \top)}{\langle \sigma, \gamma, (sn(\phi, !p) >x> \pi, \kappa, \mathsf{PS}) \rangle \rightarrow \langle \sigma, \gamma, (sn(\phi, !p)[S_{\mathcal{A}}, \top] >x> \pi, \kappa, \mathsf{PS}) \rangle}$$

$$\frac{\neg\text{ach}(\kappa, \sigma, x_o) \quad sd \in S \quad \text{match}(sn(\phi, \kappa), \sigma, sd) \quad \text{ret}(sd, x_n)}{\begin{array}{l}\langle \sigma, \gamma, (sn(\phi, \kappa)[S, x_o] >x> \pi, \kappa', \mathsf{PS}) \rangle \rightarrow \\ \qquad\qquad \langle \sigma, \gamma, (sn(\phi, \kappa)[S \setminus \{sd\}, x_n] >x> \pi, \kappa', \mathsf{PS}) \rangle\end{array}}$$

**Definition 25** *(revision function)* The revision function $rev$ is defined as follows: $rev(np, \sigma, x) = \sigma$ and $rev(p, \sigma, x) = brev(\sigma, x)$.

**Definition 26** *(goal of service call achieved after service execution)*

$$\frac{\neg\text{ach}(\kappa', \sigma, x') \quad \text{ach}(\kappa, \sigma, x') \quad rev(r, \sigma, x') = \sigma' \quad \gamma' = \gamma \setminus \{\kappa \mid \text{ach}(\kappa, \sigma, x')\}}{\langle \sigma, \gamma, (sn^r(\phi, \kappa)[S, x'] >x> \pi, \kappa', \mathsf{PS}) \rangle \rightarrow \langle \sigma', \gamma', ([x'/x]\pi, \kappa', \mathsf{PS}) \rangle}$$

**Definition 27** *(base)* The predicate $\text{base}(\sigma, \kappa, x)$ has a belief base $\sigma$, a goal $\kappa$, and a formula $x$ representing the base of $\kappa$ in $\sigma$ as parameters. Let $\sigma' \subseteq \sigma$ such that $\sigma' \models p$ and for any $\sigma''$ such that $\sigma'' \subset \sigma'$, we have $\sigma'' \not\models p$. The predicate is then defined as follows:

$$\begin{aligned}&\text{base}(\sigma, !p, x) \;\Leftrightarrow \sigma' \text{ exists and } x = \textstyle\bigwedge_{\phi \in \sigma'} \phi\\ &\text{base}(\sigma, ?p, x) \Leftrightarrow x = \bot\end{aligned}$$

**Definition 28** *(goal of service call achieved before services are called)*

$$\frac{\text{ach}(!p, \sigma, \top) \quad \text{base}(\sigma, !p, x')}{\langle \sigma, \gamma, (sn(\phi, !p) >x> \pi, \kappa', \mathsf{PS}) \rangle \rightarrow \langle \sigma, \gamma, ([x'/x]\pi, \kappa', \mathsf{PS}) \rangle}$$

**Definition 29** *(apply rule to create stack element)* Below, $\mathsf{PS}' = \mathsf{PS}_{\mathcal{A}} \setminus \{\kappa' \mid \beta \Rightarrow \pi\}$.

$$\frac{\kappa' \mid \beta \Rightarrow \pi \in \mathsf{PS}_{\mathcal{A}} \quad \text{applicable}(\kappa' \mid \beta \Rightarrow \pi, \kappa, \sigma)}{\langle \sigma, \gamma, (\kappa >x> \pi', \kappa'', \mathsf{PS}) \rangle \rightarrow \langle \sigma, \gamma, (\pi, \kappa, \mathsf{PS}').(\kappa >x> \pi', \kappa'', \mathsf{PS}) \rangle}$$

**Definition 30** *(apply rule after plan failure)* Below, $\mathsf{PS}' = \mathsf{PS} \setminus \{\kappa' \mid \beta \Rightarrow \pi\}$.

$$\frac{\kappa' \mid \beta \Rightarrow \pi \in \mathsf{PS} \quad \text{applicable}(\kappa' \mid \beta \Rightarrow \pi, \kappa, \sigma)}{\langle \sigma, \gamma, (\epsilon, \kappa, \mathsf{PS}) \rangle \rightarrow \langle \sigma, \gamma, (\pi, \kappa, \mathsf{PS}') \rangle}$$

**Definition 31** *(popping a stack element: goal of stack element reached or unreachable)*

$$\frac{\text{ach}(\kappa_1, \sigma, x) \quad rev(r, \sigma, x) = \sigma' \quad \gamma' = \gamma \setminus \{\kappa \mid \text{ach}(\kappa, \sigma', x)\}}{\begin{array}{r}\langle \sigma, \gamma, (sn_1^r(\phi_1, \kappa_3)[S, x] >x_1> \pi_1, \kappa_1, \mathsf{PS}_1).(\kappa_1 >x_2> \pi_2, \kappa_2, \mathsf{PS}_2) \rangle \rightarrow \\ \langle \sigma', \gamma', ([x/x_2]\pi_2, \kappa_2, \mathsf{PS}_2) \rangle\end{array}}$$

$$\frac{\mathcal{T}(\sigma, a) = \sigma' \quad \text{ach}(!p, \sigma', \top) \quad \text{base}(\sigma', !p, x') \quad \gamma' = \gamma \setminus \{\kappa \mid \text{ach}(\kappa, \sigma', \top)\}}{\langle \sigma, \gamma, (a \gg \pi', !p, \mathsf{PS}').(!p >x> \pi, \kappa, \mathsf{PS}) \rangle \rightarrow \quad \langle \sigma', \gamma', ([x'/x]\pi, \kappa, \mathsf{PS}) \rangle}$$

$$\frac{\neg \exists \rho \in \mathsf{PS} : \text{applicable}(\rho, \kappa, \sigma)}{\langle \sigma, \gamma, (\epsilon, \kappa, \mathsf{PS}).(\kappa >x> \pi, \kappa', \mathsf{PS}') \rangle \rightarrow \langle \sigma, \gamma, (\epsilon, \kappa', \mathsf{PS}') \rangle}$$

**Definition 32** *(plan failure)*

$$\frac{\neg \text{ach}(\kappa', \sigma, x_o) \quad \neg \exists sd \in S : \text{match}(sn(\phi, \kappa'), \sigma, sd)}{\langle \sigma, \gamma, (sn^r(\phi, \kappa')[S, x_o] >x> \pi, \kappa, \mathsf{PS}) \rangle \rightarrow \langle \sigma, \gamma, (\epsilon, \kappa, \mathsf{PS}) \rangle}$$

$$\frac{\mathcal{T}(\sigma, a) \text{ is undefined}}{\langle \sigma, \gamma, (a \gg \pi, \kappa, \mathsf{PS}) \rangle \rightarrow \langle \sigma, \gamma, (\epsilon, \kappa, \mathsf{PS}) \rangle}$$

## B Procedural Orchestration Language

The syntax and semantics of service descriptions is the same as was defined for the goal-oriented orchestration language.

### B.1 Syntax

**Definition 33** *(statement)* Assume a set InternalAction of internal actions with typical element $a$, and a set of variable names Var with typical element $x$. The set Var contains the reserved variable names *from* and *this*. Let $sn \in N_{sn}$,

$r \in \{np, p\}$, let $e$ be an exception name, let $p$ be an atom, and let $\phi$ propositional formula. Then the set of statements $\mathsf{Plan}$ with typical element $\pi$ is defined as follows, where $t$ stands for test, and $b$ stands for basic plan element.

$$\begin{array}{ll}
\kappa & ::= ?p \mid !p \\
v & ::= \mathit{true} \mid \mathit{false} \mid \phi \mid \kappa \\
t & ::= \phi? \mid (x = v)? \mid \mathit{ach}(\mathit{act}_\kappa, x)? \mid \mathit{not}\ t \mid t \wedge t' \\
\mathit{act}_i & ::= x \mid v \\
\mathit{act}_\phi & ::= x \mid \phi \\
\mathit{act}_\kappa & ::= x \mid \kappa \\
\mathit{exp} & ::= v \mid \kappa(\mathit{act}_1, \ldots, \mathit{act}_n) \mid \mathit{sn}^r(\mathit{act}_\phi, \mathit{act}_\kappa) \mid \mathit{base}(\mathit{act}_\kappa) \\
b & ::= a \mid x := \mathit{exp} \mid t \mid \mathsf{return}\ \mathit{act} \mid \mathsf{throw}\ e \\
\pi & ::= b \mid b; \pi \mid \pi + \pi' \mid \mathtt{while}\ t\ \mathtt{do}\ \pi\ \mathtt{od}
\end{array}$$

We abbreviate $\mathit{ach}(\kappa, \top)$ as $\mathit{ach}(\kappa)$.

**Definition 34** *(procedures and exception handlers)* A procedure has the form $\kappa(x_1, \ldots, x_n) \Rightarrow \pi$. Exception handlers, typically denoted by $h$, have the form $e.\mathsf{Handler} \Rightarrow \pi$, where $e$ is an exception name. A procedure definition is a procedure accompanied with a possibly empty set of exception handlers, denoted by $[\kappa \Rightarrow \pi, H]$, where $H$ is a set of exception handlers.

**Definition 35** *(procedural program)* A procedural program is a tuple $\langle \sigma_0, \gamma_0, \pi_0, \mathsf{P}, \mathcal{T} \rangle$, where the belief base $\sigma_0$ and goal base $\gamma_0$ are as in Definition 1, and together form the global state of the program, $\pi_0$ is the initial statement, $\mathsf{P}$ is a set of procedure definitions, and $\mathcal{T}$ is as in Definition 18. The initial configuration of this program is $\langle \sigma_0, \gamma_0, (\pi_0, \emptyset, \emptyset), \mathsf{P}, \mathcal{T} \rangle$.

## B.2 Semantics

**Definition 36** *(stack execution)* Let $st \neq E$.

$$\frac{\langle \sigma, \gamma, \mathrm{st} \rangle \rightsquigarrow \langle \sigma', \gamma', \mathrm{st}' \rangle}{\langle \sigma, \gamma, \mathrm{st}.\mathrm{St} \rangle \rightsquigarrow \langle \sigma', \gamma', \mathrm{st}'.\mathrm{St} \rangle}$$

**Definition 37** *(action execution)*

$$\frac{\mathcal{T}(\sigma, a) \text{ is undefined} \qquad [\kappa/\mathit{this}] \in \theta}{\langle \sigma, \gamma, (a; \pi, \theta, H) \rangle \rightsquigarrow \langle \sigma, \gamma, ([\alpha_{\mathit{fail}}]\ \mathsf{throw}\ \kappa.\mathsf{planFailedExc}, \theta, H) \rangle}$$

**Definition 38** *(assignment of values to variables)* We define $[v/x]\theta$ as $\theta \cup \{[v/x]\}$ if there is no $v'$ such that $[v'/x] \in \theta$, and as $(\theta \setminus \{[v'/x]\}) \cup \{[v/x]\}$ otherwise.

$$\frac{}{\langle \sigma, \gamma, (x := v; \pi, \theta, H) \rangle \rightsquigarrow \langle \sigma, \gamma, (\pi, [v/x]\theta, H) \rangle}$$

**Definition 39** *(procedure call)*

$$\frac{[\kappa'(f_1, \ldots, f_n) \Rightarrow \pi', H'] \text{ is a procedure definition} \qquad [v_1/x_1], \ldots, [v_m/x_m] \in \theta}{\begin{array}{l}\langle \sigma, \gamma, (x := \kappa'(x_1, \ldots, x_m, v_{m+1}, \ldots, v_n); \pi, \theta, H) \rangle \rightsquigarrow \\ \qquad\qquad \langle \sigma, \gamma, (\pi', \{[v_1/f_1], \ldots, [v_n/f_n]\}, H').(x := \kappa'; \pi, \theta, H) \rangle\end{array}}$$

**Definition 40** *(service call)*

$$\frac{sd \in S \quad \text{match}(sn(\phi,\kappa),\sigma,sd) \quad \text{ret}(sd,v)}{\langle \sigma,\gamma,(x := sn^r(\phi, act_\kappa);\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,(\pi,[v/x]\theta,H)\rangle}$$

$$\frac{\neg\exists sd \in S : \text{match}(sn(\phi,\kappa),\sigma,sd)}{\langle \sigma,\gamma,(x := sn^r(\phi,\kappa);\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,(\pi,[nomatch/x]\theta,H)\rangle}$$

If (some of) the parameters of the service call are variables, the values of these variables are first retrieved from $\theta$.

**Definition 41** *(retrieving the base)* The semantics of the predicate $base(\sigma,\kappa,v)$ is specified in Definition 27.

$$\frac{base(\sigma,\kappa,v)}{\langle \sigma,\gamma,(x := base(\kappa);\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,(\pi,[v/x]\theta,H)\rangle}$$

If the parameter of *base* is a variable, the value of this variable is first retrieved from $\theta$.

**Definition 42** *(tests)* The semantics of $ach(\kappa,\sigma,x)$ is specified in Definition 21, and the semantics of negation and conjunction is as usual.

$$\frac{\sigma \models \phi}{\langle \sigma,\gamma,(\phi?;\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,(\pi,\theta,H)\rangle} \qquad \frac{[v/x] \in \theta}{\langle \sigma,\gamma,((x=v)?;\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,(\pi,\theta,H)\rangle}$$

$$\frac{ach(\kappa,\sigma,x)}{\langle \sigma,\gamma,(ach(\kappa,x)?;\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,(\pi,\theta,H)\rangle}$$

If the first parameter of *ach* is a variable, the value of this variable is first retrieved from $\theta$.

**Definition 43** *(return)*

$$\frac{[v/x'] \in \theta'}{\langle \sigma,\gamma,(\textsf{return } x';\pi',\theta',H').(x := \kappa';\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,(\pi,[v/x]\theta,H)\rangle}$$

**Definition 44** *(throwing exceptions)*

$$\frac{e.\textsf{Handler} \Rightarrow \pi' \in H}{\langle \sigma,\gamma,(\textsf{throw } e;\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,(\pi',\theta,H)\rangle}$$

$$\frac{\neg\exists h' \in H' : h' \text{ is of the form } e.\textsf{Handler} \Rightarrow \pi''}{\langle \sigma,\gamma,(\textsf{throw } e;\pi',\theta',H').(\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,(\textsf{throw } e,\theta,H)\rangle}$$

**Definition 45** *(non-deterministic choice)*

$$\frac{\langle \sigma,\gamma,(\pi_1;\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,S\rangle}{\langle \sigma,\gamma,((\pi_1+\pi_2);\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,S\rangle} \qquad \frac{\langle \sigma,\gamma,(\pi_2;\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,S\rangle}{\langle \sigma,\gamma,((\pi_1+\pi_2);\pi,\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,S\rangle}$$

**Definition 46** *(while)* The specification of whether test $t$ holds, is given in Definition 42.

$$\frac{t \text{ holds}}{\langle \sigma,\gamma,(\texttt{while } t \texttt{ do } \pi \texttt{ od};\pi',\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,(\pi;\texttt{while } t \texttt{ do } \pi \texttt{ od};\pi',\theta,H)\rangle}$$

$$\frac{t \text{ does not hold}}{\langle \sigma,\gamma,(\texttt{while } t \texttt{ do } \pi \texttt{ od};\pi',\theta,H)\rangle \rightsquigarrow \langle \sigma,\gamma,(\pi;\texttt{while } t \texttt{ do } \pi \texttt{ od};\pi',\theta,H)\rangle}$$

# C Translation and Correctness

## C.1 Translation

**Definition 47** *(program points)* We extend statements in the procedural orchestration language with program points $\alpha$, which can be placed before any atomic statement $b$, yielding $[\alpha]b$, in a statement $\pi$. Program points do not influence the semantics, and they disappear if their corresponding atomic statement is executed. An empty statement has by definition a program point $[\alpha_\epsilon]$. We say that a configuration $d$ of a procedural program is at program point $\alpha$, if the statement of the top element of the stack of $d$ is of the form $[\alpha]\pi$, and denote this by $\alpha : d$. We define the relation $\Rightarrow$ as follows. If $\alpha_0 : d_0 \rightsquigarrow \ldots \rightsquigarrow \alpha_n : d_n$ such that no configuration $d_i$ with $0 < i < n$ is at a program point, then $\alpha_0 : d_0 \Rightarrow \alpha_n : d_n$.

**Definition 48** *(translating plan selection rules)* Without loss of generality, assume that variables in the goal-oriented orchestration language are not the reserved variables $tried_i$. Let $\mathsf{PS}$ be a set of plan selection rules. Let $\mathsf{PS}_\kappa$ be defined as $\{\kappa \mid \beta \Rightarrow \pi \ : \ \kappa \mid \beta \Rightarrow \pi \in \mathsf{PS}\}$ and let $n = \mid \mathsf{PS}_\kappa \mid$. We assume an ordering on the elements of $\mathsf{PS}_\kappa$ as follows: $\{\kappa \mid \beta_1 \Rightarrow \pi_1, \ldots, \kappa \mid \beta_n \Rightarrow \pi_n\}$. The translation function $t(\mathsf{PS}_\kappa)$ for translating $\mathsf{PS}_\kappa$ into one procedure definition is defined as follows.

$$
\begin{array}{l}
[\kappa(tried_1, \ldots, tried_n, from) \Rightarrow \\
\quad this := \kappa; \\
\quad (+_{1 \le i \le n}((tried_i = false)? \wedge \beta_i?; tried_i := true; u_\kappa(\pi_i); [\alpha_{fail}]\ \mathsf{throw}\ \kappa.\mathsf{planFailedExc}) + \\
\quad (not \bigwedge_{1 \le i \le n}((tried_i = false)? \wedge \beta_i?); [\alpha_{fail}]\ \mathsf{throw}\ from.\mathsf{planFailedExc})), \\
\{\kappa.\mathsf{planFailedExc.Handler} \Rightarrow x_f := \kappa(tried_1, \ldots, tried_n, from); \mathsf{return}\ x_f\}]
\end{array}
$$

**Definition 49** *(translating plans to statements)* We define a function $u_\kappa(\pi)$ where $\kappa$ is the head of the plan selection rule of which the body $\pi$ is translated, or the goal of the stack element containing $\pi$. Let $\mathsf{PS}_{\kappa'} = \{\kappa' \mid \beta' \Rightarrow \pi' \ : \ \kappa' \mid \beta' \Rightarrow \pi' \in \mathsf{PS}\}$, let $n' = \mid \mathsf{PS}_{\kappa'} \mid$, let $false_{1,\ldots,n'}$ be a vector of length $n'$ of parameters being the value $false$, and let $S_\mathcal{O}$ be the set of available service descriptions, and let $sd_{sn}$ be the service description of the service called for service call $sn^r(act_\phi, act_{\kappa'})$.

$$
\begin{array}{ll}
u_\kappa(\kappa' >x> \pi) & = [\alpha_{\kappa'}]\ ((ach(\kappa')?; x := base(\kappa')) + \\
 & \quad (not\ ach(\kappa')?; x := \kappa'(false_{1,\ldots,n'}, \kappa))); u_\kappa(\pi) \\
u_\kappa(a \gg \pi) & = [\alpha_a]\ a; ((ach(\kappa)?; x := base(\kappa); \mathsf{return}\ x) + \\
 & \quad (not\ ach(\kappa)?; u_\kappa(\pi))) \\
u_\kappa(sn^r(act_\phi, act_{\kappa'}) >x> \pi) & = [\alpha_{sn1}]\ x := base(act_{\kappa'}); ((ach(act_{\kappa'}, x); u_\kappa(\pi)) + \\
 & \quad (not\ ach(act_{\kappa'}, x)?; S := S_\mathcal{O}; \\
 & \quad \mathtt{while}\ not\ ach(act_{\kappa'}, x)\ \mathsf{do}\ [\alpha_{sn2}]\ x := sn^r(act_\phi, act_{\kappa'}); \\
 & \quad ((x = nomatch)?; [\alpha_{fail}]\ \mathsf{throw}\ \kappa.\mathsf{planFailedExc}) + \\
 & \quad (not(x = nomatch)?; S := S \setminus \{sd_{sn}\})\ \mathsf{od}); \\
 & \quad [\alpha_{sn3}]\ ((ach(\kappa, x)?; \mathsf{return}\ x) + (not\ ach(\kappa, x)?; u_\kappa(\pi)))
\end{array}
$$

**Definition 50** *(translating agents)* Let $\langle \sigma_0, \gamma_0, \mathsf{PS}, \mathcal{T} \rangle$ be a program in the goal-oriented orchestration language. We then define the procedure $init()$ and its exception handlers as follows, relative to goal base $\gamma_0 = \{\kappa_0, \ldots, \kappa_n\}$, with $0 \leq i \leq n$, $n_i = |\,\mathsf{PS}_{\kappa_i}\,|$, and $*$ is the disjunction of all guards of all rules in $\mathsf{PS}$.

$$[\mathit{init}() \Rightarrow +_i(\mathit{goal}(\kappa_i)?; x := \kappa_i(\mathit{false}_{1,\ldots,n_i}, \kappa_i); \mathit{stop} := \mathit{init}(); \mathsf{return}\ \mathit{stop}) + \\ ((\textstyle\bigwedge_i \mathit{not\ goal}(\kappa_i))?; \mathit{stop} := \mathit{st}; \mathsf{return}\ \mathit{stop}) + ((\mathit{not}\ *)?; \mathit{stop} := \mathit{st}; \mathsf{return}\ \mathit{stop}),$$

$$\textstyle\bigcup_i \kappa_i.\mathsf{planFailedExc} \Rightarrow \mathit{stop} := \mathit{init}(); \mathsf{return}\ \mathit{stop}]$$

The function $v$, which takes a program in the goal-oriented orchestration language and yields the corresponding program in the procedural orchestration language, is then defined as follows: $v(\langle \sigma_0, \gamma_0, \mathsf{PS}, \mathcal{T}\rangle) = \langle \sigma_0, \gamma_0, \pi_0, \mathsf{P}, \mathcal{T}\rangle$ where $\pi_0 = [\alpha_{init}]\mathit{stop} := \mathit{init}()$ and $\mathsf{P} = \bigcup_{\kappa \in \{\kappa\ :\ \kappa | \beta \Rightarrow \pi \in \mathsf{PS}\}} t(\mathsf{PS}_\kappa) \cup \{\mathit{init}()\}$ where $\mathit{init}()$ denotes the procedure definition above.

Note that we do not record which rules have been tried for achieving goals in the goal base. Goals in the goal base should not be dropped if all rules have been tried. Rather, it should be tried again to achieve these goals, if a first try failed. This level of commitment to top-level goals is standard in goal-oriented programming.

**Definition 51** *(translating statements to plans)* The function $u_\theta^{-1}$, which takes a statement at a program point from the procedural language and yields a plan in the goal-oriented language is defined as follows, where $\pi$ is a statement and $[\kappa/\mathit{this}] \in \theta$.

$$\begin{array}{ll} u_\theta^{-1}(\pi) & = u_\kappa^{-1}(\pi) \text{ for } \pi \text{ of the form of the statements of Definition 9} \\ u_\theta^{-1}(\mathtt{fail}) & = \epsilon \\ u_\theta^{-1}(\mathtt{sn{-}while}) & = \mathit{sn}^r(\mathit{act}_\phi, \mathit{act}_{\kappa'})[S_n, x_n] >x> \pi \text{ with } [S_n/S], [x_n/x] \in \theta \\ u_\theta^{-1}(\mathtt{sn-done}) & = \mathit{sn}^r(\mathit{act}_\phi, \mathit{act}_{\kappa'})[S_n, x_n] >x> \pi \text{ with } [S_n/S], [x_n/x] \in \theta \end{array}$$

where `fail` is shorthand for $[\alpha_{fail}]$ $\mathsf{throw}\ \kappa.\mathsf{planFailedExc}$, `sn−while` is shorthand for

$[\alpha_{sn2}]\ x := \mathit{sn}^r(\mathit{act}_\phi, \mathit{act}_{\kappa'}); (((x = \mathit{nomatch})?; [\alpha_{fail}]\ \mathsf{throw}\ \kappa.\mathsf{planFailedExc}) +$
$(\mathit{not}(x = \mathit{nomatch})?; S := S \setminus \{\mathit{sd}_{sn}\}));$
$\mathtt{while}\ \mathit{not\ ach}(\mathit{act}_{\kappa'}, x)\ \mathtt{do}\ [\alpha_{sn2}]\ x := \mathit{sn}^r(\mathit{act}_\phi, \mathit{act}_{\kappa'});$
$(((x = \mathit{nomatch})?; [\alpha_{fail}]\ \mathsf{throw}\ \kappa.\mathsf{planFailedExc}) +$
$(\mathit{not}(x = \mathit{nomatch})?; S := S \setminus \{\mathit{sd}_{sn}\}))\ \mathsf{od};$
$[\alpha_{sn3}]\ ((\mathit{ach}(\kappa, x)?; \mathsf{return}\ x) + (\mathit{not\ ach}(\kappa, x)?; u_\kappa(\pi)))$

and `sn − done` is shorthand for

$$[\alpha_{sn3}]\ ((\mathit{ach}(\kappa, x)?; \mathsf{return}\ x) + (\mathit{not\ ach}(\kappa, x)?; u_\kappa(\pi))).$$

**Definition 52** *(translating procedural stacks to goal-oriented stacks)* Let $\mathcal{A} = \langle \sigma_0, \gamma_0, \mathsf{PS}^\mathcal{A}, \mathcal{T}\rangle$ be a program in the goal-oriented orchestration language. We define a function $y'(\mathsf{PS}^\mathcal{A}, \theta)$, which takes a set of plan selection rules and a substitution, and which yields a set of plan selection rules. Let $[\kappa/\mathit{this}] \in \theta$ and let $\mathsf{PS}_\kappa^\mathcal{A}$ be as in Definition 48 with $n = |\,\mathsf{PS}_\kappa^\mathcal{A}\,|$ and rules numbered 1 to $n$. We then define $y'(\mathsf{PS}_\mathcal{A}, \theta)$ as

$$\{\kappa \mid \beta_i \Rightarrow \pi_i\ :\ [\mathit{false}/\mathit{tried}_i] \in \theta,\ \kappa \mid \beta_i \Rightarrow \pi_i \in |\,\mathsf{PS}_\kappa^\mathcal{A}\,|\} \cup (\mathsf{PS}^\mathcal{A} \setminus \mathsf{PS}_\kappa^\mathcal{A})$$

where $1 \leq i \leq n$.

If $d_0 \rightsquigarrow^* \alpha : d_n$ where $d_0$ is the initial configuration of $v(\mathcal{A})$, we have that the top elements of the stack $S$ of $d_n$ are of the form

$$\begin{aligned}([\alpha]\pi, \theta, H).(x_f := \kappa(\mathit{tried}_1, \ldots, \mathit{tried}_n, \mathit{from}); \mathsf{return}\ x_f, \theta_{m-1}, H).\ldots.\\ (x_f := \kappa(\mathit{tried}_1, \ldots, \mathit{tried}_n, \mathit{from}); \mathsf{return}\ x_f, \theta_1, H)\end{aligned} \tag{10}$$

where $[\kappa/\mathit{this}] \in \theta$, $n = |\,\mathsf{PS}^{\mathcal{A}}_{\kappa}\,|$, and $m = |\,\{[\mathit{true}/\mathit{tried}_i] \mid [\mathit{true}/\mathit{tried}_i] \in \theta\}\,|$ for $1 \leq i \leq n$. The elements just below these top elements are of the form

$$\begin{aligned}&(x := \kappa(\mathit{false}_1, \ldots, \mathit{false}_n, \kappa'); [\alpha']\pi', \theta', H').\\ &(x_f := \kappa'(\mathit{tried}_1, \ldots, \mathit{tried}_k, \mathit{from}); \mathsf{return}\ x_f, \theta'_{k-1}, H').\ldots.\\ &(x_f := \kappa'(\mathit{tried}_1, \ldots, \mathit{tried}_k, \mathit{from}); \mathsf{return}\ x_f, \theta'_1, H')\end{aligned} \tag{11}$$

where $[\kappa'/\mathit{this}] \in \theta'$. The first element thus has a procedure call at the start of its statement, just before reaching a program point. This pattern repeats itself, until reaching the bottom elements of $S$, which are of the form

$$\begin{aligned}&S'.(x := \kappa(\mathit{false}_{1,\ldots,n}, \kappa); \mathit{stop} := \mathit{init}(); \mathsf{return}\ \mathit{stop}, \emptyset, H).\\ &(\mathit{stop} := \mathit{init}(); \mathsf{return}\ \mathit{stop}, \emptyset, H)_1.\\ &\ldots\\ &(\mathit{stop} := \mathit{init}(); \mathsf{return}\ \mathit{stop}, \emptyset, H)_k.\\ &[\alpha_{\mathit{init}}](\mathit{stop} := \mathit{init}(), \emptyset, \emptyset)\end{aligned} \tag{12}$$

where $k \geq 0$.

We now define the function $y(S)$, which takes a stack from a procedural configuration $d_n$ as defined above, and yields a stack in the goal-oriented language, as follows, where the top elements of $S$ are of the form (10) (denoted by *top*), i.e., the top element is of the form $([\alpha]\pi, \theta, H)$ with $[\kappa/\mathit{this}] \in \theta$, *middle* is of the form (11), *init* is of the form (12), $u_\theta^{-1}(\pi)\theta$ expresses the application of $\theta$ to $u_\theta^{-1}(\pi)$, and $\pi'$, $\theta'$, and $\kappa'$ are as in (11):

$$\begin{aligned}y(\mathit{top}.S) &= (u_\theta^{-1}(\pi)\theta, \kappa, y'(\mathsf{PS}^{\mathcal{A}}, \theta)).y(S)\\ y(\mathit{middle}.S) &= (\kappa >x> u_{\theta}^{-1}(\pi')\theta', \kappa', y'(\mathsf{PS}^{\mathcal{A}}, \theta')).y(S)\\ y(\mathit{init}) &= E\\ y(([\alpha_{\mathit{init}}]\mathit{stop} := \mathit{init}(), \emptyset, \emptyset)) &= E.\end{aligned}$$

A stack $S.E$ is identified with $S$.

**Definition 53** *(translating configurations)* Let $d_n$ be a configuration in the procedural orchestration language such that $d_0 \rightsquigarrow^* \alpha : d_n$ where $d_0$ is the initial configuration of $v(\mathcal{A})$ where $\mathcal{A}$ is some program in the goal-oriented orchestration language. The function $z$ takes such a configuration $d_n$ of the form $\langle \sigma, \gamma, S \rangle$ and yields the corresponding configuration in the goal-oriented orchestration language.

$$z(\langle \sigma, \gamma, S \rangle) = \langle \sigma, \gamma, y(S) \rangle$$

## C.2 Correctness

**Lemma 1** Let $\mathcal{A}$ be a program in the goal-oriented orchestration language with initial configuration $c_0$ and $v(\mathcal{A})$ the translation of $\mathcal{A}$ with initial configuration $d_0$. Then it holds that $z(d_0) = c_0$.

*Proof:* Let $\mathcal{A} = \langle \sigma_0, \gamma_0, \mathsf{PS}, \mathcal{T} \rangle$ with $c_0 = \langle \sigma_0, \gamma_0, E, \mathsf{PS}, \mathcal{T} \rangle$ (Definition 18). We then have that $v(\mathcal{A}) = \langle \sigma_0, \gamma_0, \pi_0, \mathsf{P}, \mathcal{T} \rangle$ where $\pi_0 = [\alpha_{init}] \mathit{stop} := \mathit{init}()$ and $\mathsf{P} = \bigcup_{\kappa \in \{\kappa \,:\, \kappa | \beta \Rightarrow \pi \in \mathsf{PS}\}} t(\mathsf{PS}_\kappa) \cup \{\mathit{init}()\}$ (Definition 50). Then $v(\mathcal{A})$ has initial configuration $d_0 = \langle \sigma_0, \gamma_0, (\pi_0, \emptyset, \emptyset) \rangle$ (Definition 35). We have that $z(d_0) = \langle \sigma_0, \gamma_0, E \rangle$ by Definition 53, which concludes the proof. □

**Lemma 2** Let $P$ be a program in the procedural orchestration language with initial configuration $d_0$ such that there is some program $\mathcal{A}$ of the goal-oriented orchestration language with $v(\mathcal{A}) = P$ and let $d_0 \rightsquigarrow d_1 \rightsquigarrow \ldots$ be a run where $z(d_0) = c_0$ for $c_0$ being the initial configuration of $\mathcal{A}$. Let $0 = p_0 < p_1 < \ldots$ be indices such that for each $d_{p_i}$ with $i \geq 0$, $d_{p_i}$ is at a program point and $\alpha_i : d_{p_i} \Rightarrow \alpha_{i+1} : d_{p_{i+1}}$ holds. We then have that for each $d_{p_i}$ it holds that $z(d_{p_i})$ is defined.

*Proof:* Definition 53 translates procedural configurations by translating their stack, as specified in Definition 52 through the function $y$. The function $y$ specifies how to translate a stack at program point $\alpha_{init}$. The translation of all other program points is specified through the function $u_\theta^{-1}$ (Definition 51), which is used by $y$. □

**Theorem 2** Let $\mathcal{A}$ be a program in the goal-oriented orchestration language with initial configuration $c_0$ and $v(\mathcal{A})$ the translation of $\mathcal{A}$. Then it holds for any run $c_0 \rightarrow c_1 \rightarrow \ldots$ that there exist indices $0 = p_0 < p_1 < \ldots$ and configurations $d_0, d_1, \ldots$ such that $d_0 \rightsquigarrow d_1 \rightsquigarrow \ldots$ is a run in the procedural orchestration language, $d_0$ is the initial configuration of $v(\mathcal{A})$, and for all $p_i$ with $i \geq 0$ it holds that $z(d_{p_i}) = c_i$.

Let $P$ be a program in the procedural orchestration language with initial configuration $d_0$ such that there is some program $\mathcal{A}$ of the goal-oriented orchestration language with $v(\mathcal{A}) = P$. Then it holds for any run $d_0 \rightsquigarrow d_1 \rightsquigarrow \ldots$ that there exist indices $0 = p_0 < p_1 < \ldots$ and configurations $c_0, c_1, \ldots$, such that $c_0 \rightarrow c_1 \rightarrow \ldots$ is a run in the goal-oriented orchestration language, $c_0$ is the initial configuration of $\mathcal{A}$, and for all $p_i$ with $i \geq 0$, it holds that $z(d_{p_i}) = c_i$.

*Proof:* Let $\mathcal{A}$ be a program in the goal-oriented orchestration language with initial configuration $c_0$ and $v(\mathcal{A})$ the translation of $\mathcal{A}$.

**Part 1** Let $c_0 \rightarrow \ldots c_1 \rightarrow \ldots$ be an arbitrary run and let $d_0$ be the initial configuration of $v(\mathcal{A})$. We prove the result by induction over the length of runs in the goal-oriented orchestration language, and by structural induction (reasoning by cases) to prove the result for arbitrary configurations. We have to show that there exist indices $0 = p_0 < p_1 < \ldots$ and configurations $d_0, d_1, \ldots$ such that $d_0 \rightsquigarrow d_1 \rightsquigarrow \ldots$ is a run in the procedural orchestration language, and for all $p_i$ with $i \geq 0$ it holds that $z(d_{p_i}) = c_i$.

We show that if $z(\alpha_i : d_{p_i}) = c_i$ for $i \geq 0$, there is a $p_{i+1}$ such that $\alpha_i : d_{p_i} \Rightarrow \alpha_{i+1} : d_{p_{i+1}}$ and $z(\alpha_{i+1} : d_{p_{i+1}}) = c_{i+1}$. We have that $z(\alpha_i : d_{p_i})$ is defined by Lemma 2. Further, we have that $z(\alpha_{init} : d_0) = c_0$ by Lemma 1, which provides the induction basis.

For the induction step, we only show the case where the plan of the top element of the stack of $c_i$ is a subgoal, i.e., where $c_i$ is of the form $\langle \sigma, \gamma, (\kappa' > x > \pi, \kappa, \mathsf{PS}).S \rangle$. Other cases are analogous. All configurations $\alpha : d_i$ from the procedural run for which it holds that $z(\alpha_i : d_{p_i}) = c_i$, have the following form (Definition 52):

$$\begin{aligned}\langle \sigma, \gamma, ([\alpha_{\kappa'}]\pi_{proc}, \theta, H).(x_f := \kappa(\mathit{tried}_1, \ldots, \mathit{tried}_n, \mathit{from}); \mathsf{return}\ x_f, \theta_{m-1}, H). \ldots . \\ (x_f := \kappa(\mathit{tried}_1, \ldots, \mathit{tried}_n, \mathit{from}); \mathsf{return}\ x_f, \theta_1, H).S' \rangle\end{aligned} \tag{13}$$

where $\pi_{proc}$ is of the form

$$((\mathit{ach}(\kappa')?; x := \mathit{base}(\kappa')) + (\mathit{not}\ \mathit{ach}(\kappa')?; x := \kappa'(\mathit{false}_{1,\ldots,n'}, \kappa))); \pi'$$

where $u_\theta^{-1}(\pi')\theta = \pi$, $y(S') = S$, and $y'(\mathsf{PS}^{\mathcal{A}}, \theta) = \mathsf{PS}$.

Two possible transitions are derivable from $c_i$.

**(A)** Assume that $\mathit{ach}(\kappa', \sigma, \top)$ and $\mathit{base}(\sigma, \kappa', \phi)$. Then the following transition is derivable.

$$\langle \sigma, \gamma, (\kappa' > x > \pi, \kappa, \mathsf{PS}).S \rangle \rightarrow \langle \sigma, \gamma, ([\phi/x]\pi, \kappa, \mathsf{PS}).S \rangle \tag{14}$$

From $\mathit{ach}(\kappa', \sigma, \top)$ in the goal-oriented case, we have that $\mathit{ach}(\kappa')?$ holds in the procedural case (Definition 42). The first of the two possible non-deterministic choices of $\pi$ will thus be executed. From $\mathit{base}(\sigma, \kappa', \phi)$ in the goal-oriented case we have that $x := \mathit{base}(\kappa')$ assigns $\phi$ to $x$, i.e., the substitution in the next configuration will be $[\phi/x]\theta$ (Definition 41). We are then at the next program point, as we have $u_\theta^{-1}(\pi')\theta = \pi$. Applying the function $z$ to that configuration, we get that the result is equal to $c_{i+1}$, as $u_\theta^{-1}(\pi')[\phi/x]\theta = [\phi/x]\pi$, $y(S') = S$, $[\kappa/\mathit{this}] \in \theta$, and the *tried* variables of $\theta$ have not changed, and therefore $y'(\mathsf{PS}^{\mathcal{A}}, [\phi/x]\theta) = \mathsf{PS}$.

**(B)** Assume that $\kappa' \mid \beta_1 \Rightarrow \pi_1 \in \mathsf{PS}^{\mathcal{A}}$ and that this rule is applicable, i.e., we have that $\neg \mathit{ach}(\kappa', \sigma, \top)$. Then the following transition is derivable.

$$\langle \sigma, \gamma, (\kappa' > x > \pi, \kappa, \mathsf{PS}).S \rangle \rightarrow \langle \sigma, \gamma, (\pi_1, \kappa', \mathsf{PS}').(\kappa' > x > \pi, \kappa, \mathsf{PS}).S \rangle \tag{15}$$

We have that $\alpha_i : d_{p_i}$ is of the form (13). From $\neg \mathit{ach}(\kappa', \sigma, \top)$ in the goal-oriented case, we have that *not* $\mathit{ach}(\kappa')?$ in the procedural case. The second of the two possible non-deterministic choices of $\pi$ will thus be executed. The procedure $\kappa'(\mathit{false}_{1,\ldots,n'}, \kappa)$ is then called, which leads to the creation of a new stack element with a statement of the form

$\mathit{this} := \kappa';$
$(+_{1 \leq i \leq n}((\mathit{tried}_i = \mathit{false})? \wedge \beta_i?; \mathit{tried}_i := \mathit{true}; \pi'_i; [\alpha_{fail}]\ \mathsf{throw}\ \kappa.\mathsf{planFailedExc})\ +$
$(\mathit{not} \bigwedge_{1 \leq i \leq n}((\mathit{tried}_i = \mathit{false})? \wedge \beta_i?); [\alpha_{fail}]\ \mathsf{throw}\ \mathit{from}.\mathsf{planFailedExc}))$

where the substitution $\theta'$ of this stack element is $\{[false/tried_1], \ldots, [false/tried_{n'}, [\kappa/from]\}$ and $u^1_\theta(\pi'_i)\theta = \pi_1$. The first assignment of the statement is then executed, leading to the addition of $[\kappa'/this]$ to $\theta'$. We have that the rule $\kappa' \mid \beta_1 \Rightarrow \pi_1 \in \mathsf{PS}^{\mathcal{A}}$ is applicable, which means that $\beta_1$ holds, and $tried_1 = false$. The first non-deterministic choice can thus be executed, after which the variable $tried_1$ is set to $true$. We then reach the next program point at $\pi'_i$.

If we now apply $z$ to the reached configuration. We have that $u^{-1}_\theta(\pi'_i)\theta = \pi_1$, i.e., the statement of the top element is translated correctly into $\pi_1$. Further, we have that $[\kappa'/this]$ is in the substitution of the top element, which is correctly translated into the goal of the top element of the goal-oriented stack. Also, we have that the only *tried* variable set to *true* is $tried_1$, which means that the set of plan selection rules resulting from the translation are $\mathsf{PS}$ without the applied rule $\kappa' \mid \beta_1 \Rightarrow \pi_1$, which is equal to $\mathsf{PS}'$.

The second element and the next $n$ elements of the procedural stack are of the form (13), except that the statement of the first of these elements is now of the form $x := \kappa'(false_{1,\ldots,n'}, \kappa)));\pi'$. Applying $y$ to these elements, we get that all but the first element are disregarded. The statement $x := \kappa'(false_{1,\ldots,n'}, \kappa)));\pi'$ is translated into $\kappa' >x> \pi$, as we have $u^{-1}_\theta(\pi')\theta = \pi$. The translation of the other components is similar to case (**A**), yielding the desired result.

**Part 2** Let $P$ be a program in the procedural orchestration language with initial configuration $d_0$ such that there is some program $\mathcal{A}$ of the goal-oriented orchestration language with $v(\mathcal{A}) = P$ and let $d_0 \rightsquigarrow d_1 \rightsquigarrow \ldots$ be a run of $P$. Let $0 = p_0 < p_1 < \ldots$ be indices such that for each $d_{p_i}$ with $i \geq 0$, $d_{p_i}$ is at a program point and $\alpha_i : d_{p_i} \Rightarrow \alpha_{i+1} : d_{p_{i+1}}$ holds.

We prove that for all $p_i$ with $i \geq 0$, it holds that $z(d_{p_i}) = c_i$ and $c_0 \to c_1 \to \ldots$ is a run of $\mathcal{A}$, by induction over the length of $d_{p_0} \rightsquigarrow^* d_{p_1} \rightsquigarrow^* \ldots$. We have that $p_0 = 0$, and we thus have $z(d_{p_0}) = c_0$ by Lemma 1, providing the induction basis. Let $i \geq 0$ and let $z(d_{p_i}) = c_i$. We have to show for $c_{i+1}$ with $z(d_{p_{i+1}}) = c_{i+1}$ that $c_i \to c_{i+1}$. We show the result only in case the program point of $d_{p_i}$ is a subgoal program point.

Let $d_{p_i}$ be of the form

$$\langle \sigma, \gamma, ([\alpha_{\kappa'}]\ ((ach(\kappa')?; x := base(\kappa'))+ \\ (not\ ach(\kappa')?; x := \kappa'(false_{1,\ldots,n'}, \kappa))); \pi', \theta, H).S\rangle \quad (16)$$

i.e., we have $\alpha_{\kappa'} : d_{p_i}$. We also have that $z(d_{p_i}) = \langle \sigma, \gamma, (\kappa' >x> u^{-1}_\theta(\pi')\theta, \kappa, \mathsf{PS}).y(S)\rangle$, where $\mathsf{PS}$ is obtained from $\theta$. We then have that $c_{i+1}$ is of the form of resulting configuration of the transition of (14), or of (15), respectively, where $u^{-1}_\theta(\pi')\theta = \pi$. Assume $c_{i+1}$ is of the form (14). We know that the next program point to be encountered if $d_{p_i}$ is executed and the first test of the non-deterministic choice succeeds, occurs at $\pi'$ (Definition 9), i.e., $d_{p_{i+1}}$ is in that case of the form $\langle \sigma, \gamma, ([\alpha]\pi', \theta', H).S\rangle$ where $\alpha$ is the program point of $\pi'$. We have that $z(d_{p_{i+1}}) = \langle \sigma, \gamma, (\pi\theta', \kappa, \mathsf{PS}').y(S)\rangle$. We have that $\theta' = [base(\kappa')/x]\theta$. We thus have that $\pi\theta'$ is equal to the plan of the top element of the stack of (14), and the only change to $\theta$ was with respect to the variable $x$. We can thus conclude that $\mathsf{PS}' = \mathsf{PS}$, yielding the desired result. The other case is analogous. $\square$