Goal-Oriented and Procedural Service Orchestration*

A Formal Comparison

M. Birna van Riemsdijk Martin Wirsing

Ludwig-Maximilians-Universität München, Germany
{riemsdijk, 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.

1 Introduction

In the field of agent-oriented programming, there is an increasing amount of research on the use of *goals* in agent programming languages (see, e.g., [24, 8, 21, 17, 3, 9, 20]). Goals form a *declarative description* of the desired end result of the execution of (part of) a program. They are thus comparable to postconditions as commonly used in program verification. However, the important difference between goals and postconditions is that goals, in contrast with postconditions, are *part of the program*. A goal-oriented language has language constructs which express the goal that is to be reached by some part of the program.[1]

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 [24, 19, 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.

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

[1] Goal-oriented programming should not be confused with logic programming. While the latter is in principle purely declarative, goal-oriented programming has both declarative and procedural features.

Goal-oriented programming is targeted at dynamic domains such as agent-based systems in which the programmer does not have full control over all aspects of system behavior, e.g., due to the existence of other agents or environmental aspects outside control of the agent. In such systems, one always needs to take into account that things might "go wrong". In more restricted settings in which one can prove that a program always fulfills some desired post-condition (perhaps assuming some generally valid preconditions), goal-oriented programming is superfluous (apart from possible modeling advantages of using goals). Monitoring the execution for goal achievement does not add anything in that case, as the program was already proven to satisfy the postconditions or goals.

We argue that the domain of service-oriented computing is, like agent-based systems, a domain well-suited for using goal-oriented techniques (see, e.g., [2, 12, 1, 6] for other proposals for combining agent-oriented and service-oriented approaches). In the service-oriented systems domain, services are called on the basis of service descriptions without knowing anything about the internal architecture or workings of the service. One will typically not have or be able to obtain (formal) guarantees that the service behaves as it should. In such a setting, one will thus always need to take into account that a service does not behave as expected or desired. Moreover, in such a context it is more natural than in classical settings to specify alternative plans for reaching a goal. In more classical settings such as database applications there will typically not be alternative ways of reaching a desired result, e.g., in case accessing a database fails. In service-oriented systems, on the other hand, trying alternative ways of reaching a goal is more natural. For example, if booking a ticket with Lufthansa did not succeed, one might try booking a ticket with KLM, or if booking a plane turns out not to be possible as it is too expensive, one might try booking a train.

To investigate how goal-oriented techniques can be applied in the context of service-oriented systems, we have proposed an abstract *goal-oriented orchestration language* [23] (Section 3). A natural question that arises, given that we argue that goal-oriented techniques increase flexibility in handling failures, is how this kind of failure handling compares to more standard exception handling mechanisms. The aim of this paper is to answer this question. Our approach is that we define a procedural orchestration language with an exception handling mechanism inspired by that of WS-BPEL [10] (Section 4). We then show how a program in the goal-oriented orchestration language can be translated into a program in the procedural orchestration language that has *provably* the same behavior (Section 5). We will 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 the programming patterns resulting from the translation do not increase understandability of the code.

It is important to remark that the orchestration language of [23] is not meant to be a full-fledged orchestration language. It is based on propositional logic, and is used to investigate the semantic foundations[2] of goal-oriented orchestration

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

languages. The relative simplicity of propositional logic allows us to focus on the essential aspects of such a language. This paper contributes to the investigation of the semantic foundations of goal-oriented orchestration, and hence is also based on the simple language [23]. We are currently investigating how we can replace propositional logic by other logics such as description logic, to make the language more practically useful and to facilitate more extensive experimentation with it. We refer to [5] for the description of a goal-oriented agent programming language and platform based on first-order logic rather than propositional logic, which uses similar goal-oriented techniques as the ones we use in this paper.

Moreover, we remark that this paper addresses the composition of services using orchestration languages. The idea is that the *programmer* specifies which compositions are appropriate, using the constructs of the orchestration language. At run-time, the orchestration is executed as specified. This is in contrast with approaches to service composition based on planning (see, e.g., [14]). In the latter approaches, a composition is generated *automatically* on the basis of service descriptions and a specification of desired behavior. Nevertheless, planning approaches and programming approaches have many commonalities, and can sometimes be combined [16].

2 Example: Car Breakdown

In order to illustrate our approach, we use a very simple car breakdown scenario that is adapted from the automotive case study of the SENSORIA project [25] on service-oriented computing. We have used a variant of this scenario in [23]. In the scenario, the car has a diagnostic system which reports a failure, resulting in the car no longer being drivable. The car is furthermore endowed with orchestration software that should assist the driver in getting the appropriate support by calling, e.g., a service to get road side assistance. We assume there are also services available for calling a taxi, for making garage appointments, for ordering a tow truck, and for getting technical advice over the phone (this service makes sure the driver is phoned by the appropriate technical assistant).

Using the goal-oriented orchestration language, one can specify which plan may be executed for achieving a certain goal, under certain circumstances using so-called *plan selection rules* [19]. Plans essentially consist of service calls (where the goal to be achieved through the service call is passed as a parameter, possibly together with some additional information), subgoals (which are to be achieved by selecting an appropriate plan by means of plan selection rules), and a construct for sequential composition (inspired by the orchestration language Orc [4]) that can be used for passing along the result of service calls to other service calls. Services can be called directly by specifying the service name, or they can be discovered by matching available service descriptions to the goal of a service call, i.e., through *semantic matchmaking*. Goals to be achieved are preceded by an exclamation mark.

In this example, we assume the driver is on his way to work, i.e., he has "being at work" as its top-level goal. If the car is broken, he may either leave the

car behind and call a taxi (if he is in a hurry and near to his office), or try to get the car repaired. There are three alternative plans for getting the car repaired: the driver can repair the car himself with the help of technical support over the phone (if he is a member of this service and the car is repairable on the spot), he can get road side assistance, or have the car towed to a garage (if it is not repairable on the spot). Below, we sketch the corresponding plan selection rules. Plan selection rules have the form $\kappa \mid \beta \Rightarrow \pi$, which intuitively says that the plan π can be used to reach goal κ if β is the case.

$$
\begin{array}{ll}
\text{!atWork} \mid \text{carBroken} \wedge \text{hurry} \wedge \text{nearOffice} & \Rightarrow d(!(\text{taxi} \leq 50 \text{ euro})) \gg \text{monitor}(\text{!atWork}) \\
\text{!atWork} \mid \text{carBroken} & \Rightarrow \text{!carRepaired} \gg \text{monitor}(\text{!atWork}) \\
\text{!carRepaired} \mid \text{memberTS} \wedge \text{repOnSpot} & \Rightarrow \text{techSupport}(\text{symp}, \text{!appTA}) >x> \\
 & \quad \text{notify}(x) \ldots \\
\text{!carRepaired} \mid \textit{true} & \Rightarrow d(\text{locationCar}, \text{!roadSideAss}) \ldots \\
\text{!carRepaired} \mid \neg\text{repOnSpot} & \Rightarrow \text{!appGarage} \gg \text{!appTowTruck} \ldots
\end{array}
$$

We leave out the plan selection rules for the subgoals !appGarage and !appTowTruck for reasons of space. It may be the case that multiple plan selection rules are applicable in a certain situation. For example, if the car is broken and the driver is in a hurry and near the office, either the first or the second rule may be applied. In our abstract formal framework, one applicable rule is selected non-deterministically. However, the framework may be extended with a preference ordering over the rules. If we assume the first rule is selected, it may be the case that discovering a taxi service fails (the "d" stands for "discovery"), e.g., because it was not possible to find a taxi service for less than 50 euro. The plan is then aborted and the goal of being at work has not been achieved, after which the other applicable plan selection rule will be tried, i.e., it will be tried to achieve the subgoal of getting the car repaired. Note that it is thus useful that multiple plan selection rules are applicable in a certain situation, as another plan can then be tried if one fails. The monitoring service monitors whether, e.g., taking a taxi has resulted in the goal of being at work being achieved.

In order to achieve the subgoal of getting the car repaired, one might first try to repair the car with help of technical support over the phone (passing the symptoms of the car problem to the service). If contacting the technical support service is successful, the plan continues by passing along the result to a service that notifies the secretary of the driver about this. If calling the technical support service fails, e.g., because it turned out the membership has expired, or the service could not provide satisfactory support, the plan is aborted and another plan for reaching the goal of getting the car repaired can be tried. Our goal-oriented orchestration language tries each alternative plan to achieve a certain (instance of a) subgoal *once*, in order to prevent the orchestration from getting stuck by trying the same plans over and over to reach some subgoal.

A sketch of how this example could be programmed in a procedural orchestration language is provided below (we only show the "carRepaired" part).

```
carRepaired(tried1, tried2, tried3, from) ⇒ if tried1 = false ∧ memberTS ∧ repOnSpot
  then tried1 := true; x := techSupport(symp, !appTA);
    if ¬ach(!appTA) then throw !carRepaired.planFailedExc else notify(x) ... fi
  else if tried2 = false then tried2 := true; d(locationCar, !roadSideAss);
        if ¬ach(!roadSideAss) then throw !carRepaired.planFailedExc else ... fi
      else if tried3 = false ∧ ¬repOnSpot
          then tried3 := true; appGarage(...); appTowTruck(...)
          else throw from.planFailedExc fi fi fi

!carRepaired.planFailedExc ⇒ carRepaired(tried1, tried2, tried3, from)
```

The various plan selection rules for achieving a particular goal (!carRepaired in this case) are combined into one procedure, and subgoals occurring in plans are translated into procedure calls. After each service call, it is checked whether the service call was successful in achieving its goal ($ach(goal)$). If not, an exception is thrown, as the plan should be aborted in this case. The exception handler for !carRepaired.planFailedExc as specified above calls the procedure "carRepaired" recursively, so that *another* plan can be tried to achieve the goal. We use the variables $tried_i$ to record which plans have already been tried to reach the goal. If all plans have been tried and/or none are applicable, the exception *from*.planFailedExc is thrown which is caught lower down in the procedure call stack (in the procedure "atWork" in this case, as the procedure for repairing the car will be called from there, as recorded in the variable *from*).

We believe the code of this procedural orchestration is less understandable than the goal-oriented version, and we thus argue that goal-oriented abstractions are worth considering as language constructs of an orchestration language.[3] The purpose of the rest of this paper is to analyze the failure handling mechanism of the goal-oriented orchestration language in more detail, and to investigate the relation between the goal-oriented and procedural orchestration language from a foundational perspective by showing how an arbitrary goal-oriented orchestration can be translated into a procedural orchestration that has provably the same behavior.

3 Goal-Oriented Orchestration Language

In this section, we present the syntax and informal semantics of our goal-oriented orchestration language (Section 3.1), and the part of the formal semantics that is relevant for failure handling (Section 3.2). For reasons of space, we cannot provide the full semantics. We refer to [23, 22] for more details and explanation.

[3] Albeit not necessarily to replace procedural programming constructs, but at least in addition to them.

3.1 Syntax and Informal Semantics

Most of the ingredients of the goal-oriented orchestration language have already been introduced informally in Section 2. Here, we provide the full syntax and introduce the formal notation. A program in the goal-oriented orchestration language is called an agent, which is formally a tuple $\langle \sigma_0, \gamma_0, \mathsf{PS}, \mathcal{T} \rangle$. The initial *belief base* σ_0 represents what the agent believes to be the case in the world (comparable with the state of a procedural program), and is a consistent set of propositional formulas [19]. The initial *goal base* γ_0 is the set of top-level goals of the agent. Goals are deleted from the goal base if they are believed to be achieved [19] and are typically denoted by κ. A goal can be either an achievement goal $!p$ (where p is an atom)[4], 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. Test goals are to be fulfilled by so-called information providing services, and achievement goals may be fulfilled by world altering services [13]. PS is a set of *plan selection rules*, formally denoted as $\kappa \mid \beta \Rightarrow \pi$, where β is a propositional formula representing a condition on the beliefs that should hold for the rule to be applicable, and π is a plan. The function $\mathcal{T} : (\mathsf{BasicAction} \times \Sigma) \rightarrow \Sigma$ is a partial *belief update function* (where Σ is a set of belief bases) which specifies the belief update resulting from the execution of (internal) actions by the agent. This function is introduced as usual [19] for technical convenience.

The formal definition of the syntax of plans is given below, where x is a variable name.

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

Internal actions are typically denoted by a, and κ represents a subgoal. 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 discovered), act_κ represents the goal that is to be achieved through calling the service, and act_ϕ is (or should be instantiated with) a propositional formula representing additional information that forms input to the service. The revision parameter r can be np (non-persistent), meaning that the result of the service call is not stored in the belief base, or p (persistent), meaning that the result is stored in the belief base. The result returned from a basic plan element b is bound to the variable x, which may be used in the remaining plan π. A plan of the form $b \gg \pi$ is used to abbreviate a plan $b >x> \pi$ where x does not occur in π.

The mechanism of applying plan selection rules to goals in the goal base or subgoals in plans 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 κ is the (sub)goal to which

[4] In [23], we used arbitrary propositional formulas for the representation of goals, but for reasons of simplicity we use atoms here.

the plan selection rule has been applied, π is the plan currently being executed in order to achieve κ, and PS is the set of plan selection rules that have not yet been tried in order to achieve κ.

A stack element is popped just after a service call or an action execution if the goal of the stack element is reached, or it is popped if the goal is unreachable, meaning that there are no applicable plan selection rules. In the former case the result of the service call or the part of the belief base that expresses that the subgoal κ has been reached is returned and all occurrences of x in π are substituted with this result. The latter case is explained in Section 3.2.

A configuration of a goal-oriented program has the form $\langle \sigma, \gamma, \mathrm{St}, \mathsf{PS}, \mathcal{T} \rangle$, where St is the stack. The initial configuration of an agent $\langle \sigma_0, \gamma_0, \mathsf{PS}, \mathcal{T} \rangle$ is $\langle \sigma_0, \gamma_0, E, \mathsf{PS}, \mathcal{T} \rangle$, where E denotes an empty stack. In the transition rules, we leave out PS and $\mathcal{T}$ from configurations for reasons of presentation (and these do not change during computation).

3.2 Formal Semantics of Failure Handling

The formal semantics of our goal-oriented orchestration language is defined using a transition system [15]. 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.

In the goal-oriented orchestration language, a *failure* is not only caused by abnormalities in trying to execute some operation, but also by *being unsuccessful in reaching a goal*. In particular, if a service is called and returns some result, the call is only considered to be successful if the goal of the service call is reached through the result 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 run-time. 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 not 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.[5] 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.

The latter case is specified formally in Definition 1 below. The service call construct $sn^r(\phi, \kappa')$ (we assume variables are instantiated when the service is

[5] One might argue that a comprehensive failure handling mechanism should include compensation, but this is without the scope of this paper.

called) is annotated with a set of service descriptions S which represents services that have not yet been called, and the result x_0 of the last service call. In this setting, services are assumed to return a propositional formula that expresses the effect or piece of information resulting from calling a world altering or information providing service, respectively. The predicate $\text{ach}(\kappa, \sigma, x_o)$ holds iff the goal κ is achieved with respect to belief base σ and the service call result x_0. In case κ is an achievement goal, it is achieved if the goal follows from the belief base after it is updated with x_0. In case κ is 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 the idea is that 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 σ.

Definition 1 *(plan failure)*

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

As plans are dropped if something goes wrong (if an internal action cannot be executed, the plan is dropped as well), the occurrence of an empty plan in a stack element indicates a failure. It can also be the case that a plan is *completely executed* resulting in an empty plan, without occurrence of a problem with an action execution or service call. However, this also indicates that the plan has failed to reach the goal of the stack element, as the stack element would have been popped immediately if its goal would have been reached after an action execution or service call.

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. This is formally specified by the transition rule below. Note that a plan selection rule that is applied is *removed* from the set of available plan selection rules PS. 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 2 *(apply rule after plan failure)* Below, $\mathsf{PS}' = \mathsf{PS} \setminus \{\kappa' \mid \beta \Rightarrow \pi\}$.

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

If the plan of the top stack element is empty and there are *no* plan selection rules applicable to the subgoal κ of this stack element, the subgoal is considered to have failed definitively. Then, the top element of the stack is popped, and the plan $\kappa >x> \pi$ that contains κ is dropped from the new top element. Consecutively,

the agent can try another plan for reaching the subgoal κ', or, if there are no applicable plan selection rules, the stack element with subgoal κ' is popped as well, etc.

Definition 3 *(subgoal failure)*

$$\frac{\neg\exists(\kappa \mid \beta \Rightarrow \pi) \in \mathsf{PS} : \sigma \models \beta}{\langle \sigma, \gamma, (\epsilon, \kappa, \mathsf{PS}).(\kappa >\!x\!> \pi, \kappa', \mathsf{PS}')\rangle \rightarrow \langle \sigma, \gamma, (\epsilon, \kappa', \mathsf{PS}')\rangle}$$

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 syntax of statements is formally defined below, where e is an exception name, x is a variable name, and act_ϕ, act_κ as in Section 3.1.

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

The language of procedure names is the same as the language of goals of the goal-oriented orchestration language (κ). The (global) state of configurations in this language contains a belief base as also used in the goal-oriented orchestration language. Additionally, procedures may use local variables, typically denoted by x. These local variables may have a value v, which is *true*, *false*, a string denoting a formula ϕ, or a string denoting a procedure name κ. Tests can be global tests on the belief base ϕ? (note the difference with test goals $?p$, which can only be fulfilled through service calls), local tests $(x = v)$? that can be used for testing the value of a variable, or $\mathit{ach}(act_\kappa, x)$, which tests whether the goal act_κ is achieved with respect to the value of the variable x. Expressions are values, procedure calls $\kappa(act_1, \ldots, act_n)$, service calls, or a call to a predefined function $\mathit{base}(act_\kappa)$, which returns a conjunction of formulas from the belief base from which act_κ follows, or *false* if κ does not follow. Intuitively, this represents how κ is achieved. Elementary statements can be actions to change the belief base (as in the goal-oriented orchestration language), assignments to change the value of local variables, tests, returning a variable, and throwing 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 [10]. 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.[6] 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 4 *(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 (π, θ, H) corresponds to a procedure call, where π is the statement that still needs to be executed, θ 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 σ and goal base γ (together forming the global state), a stack St, a set of procedure definitions P, and a belief update function $\mathcal{T}$. The goal base is simply 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. For reasons of space, we do not explain nor define aspects having to do with updating of the goal base in this paper. 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.

We only show the transition rules for exception handling, for reasons of space. The semantics of the other constructs is as one would expect, and for formal details we refer to [22]. 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 π' 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.

[6] Additionally, WS-BPEL has a compensation mechanism (see also [11]), which is, however, outside the scope of this paper.

Definition 5 *(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 [7] 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".[7] 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. 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.

In this paper we present the most important parts of the translation, i.e., the translation of plan selection rules and the translation of plans. For the full technical details of the translation, we refer to [22].

Definition 6 *(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 PS be a set of plan selection rules. Let PS_κ 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 PS_κ as follows: $\{\kappa \mid \beta_1 \Rightarrow \pi_1, \ldots, \kappa \mid \beta_n \Rightarrow \pi_n\}$. The translation function t for translating PS_κ into one procedure definition is defined as follows.

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

$\{\kappa.\mathsf{planFailedExc.Handler} \Rightarrow x_f := \kappa(tried_1, \ldots, tried_n, from); \mathsf{return}\ x_f\}]$

[7] 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.

The example in Section 2 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.[8] The translation of plans is specified through the function u_κ (Definition 7).

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 3). The throwing of an exception in case a service call fails is specified in Definition 7.

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 κ 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 κ was first called (as passed to κ 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 3).

We associate with each procedure κ a handler for the exception κ.planFailedExc. This handler specifies that the procedure should be called recursively with the variables $tried_i$ (representing which plans have already been tried) as parameters. This recursive call makes sure that if a plan fails, another plan is tried which has not been tried yet (Definition 2).

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. The next definition specifies the function u_κ, which translates plans of the bodies of plan selection rules with head κ into statements of the procedural language. The function is also used to translate the plan of a stack element with subgoal κ.

Definition 7 *(translating plans to statements)* We define a function $u_\kappa(\pi)$ where κ is the head of the plan selection rule of which the body π is translated, or the goal of the stack element containing π. 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*, let $S_\mathcal{O}$ be the set of available service

[8] 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.

descriptions, and let sd_{sn} be the service description of the service called for service call $sn^r(act_\phi, act_{\kappa'})$.

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

A subgoal $\kappa' >\!x\!> \pi$ is translated into a non-deterministic choice, followed by the translation of π. The non-deterministic choice expresses that if the goal κ' is already reached before calling the procedure κ', x gets a value through the function $base(\kappa')$. If κ' is not yet achieved, the procedure κ' is called, which returns a value (a propositional formula) that expresses how κ' was achieved or an exception in case κ' could not be achieved. The actual parameters for the procedure κ' are a series of *false* values, expressing that no plans have yet been tried to reach κ', and the last parameter is the subgoal κ, which is the goal to be reached through execution of the statement $u_\kappa(\kappa' >\!x\!> \pi)$ (as we are translating plan selection rules with head κ). The translation of an action a expresses that a should be executed, and, depending on whether the goal κ 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 1).

Using the translation functions as defined above, we have defined a function v (see [22] 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 6 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. Furthermore, the procedure is defined such that the program terminates if the goal base is empty.

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 transition 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 [22] for its definition), which trans-

lates 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 (using the inverse of the function u_κ). The function uses the substitution of stack elements to determine the goal of the resulting goal-oriented stack element and to determine which plan selection rules have not yet been tried to reach the goal.

The correctness of the translation is formulated formally below. We refer to [22] for the 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 $c_0 \rightarrow c_1 \rightarrow \ldots$ 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 $z(d_{p_i}) = c_i$.

6 Conclusion

In this paper, we have shown how the goal-oriented orchestration language of [23] 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.

We are currently working on 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. 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. One possible direction for future research is to investigate whether the KAOS goal-oriented requirements engineering methodology [18] can be adapted to fit the goal-oriented orchestration language.

References

1. M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Interaction protocols and capabilities: A preliminary report. In *Principles and Practice of Semantic Web Reasoning, 4th International Workshop (PPSWR'06)*, pages 63–77, 2006.
2. L. Bozzo, V. Mascardi, D. Ancona, and P. Busetta. CooWS: Adaptive BDI agents meet service-oriented programming. In *Proceedings of the IADIS International Conference WWW/Internet 2005*, volume 2, pages 205–209. IADIS Press, 2005.
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. 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.
5. 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.
6. I. Dickinson and M. Wooldridge. Agents are not (just) web services: considering BDI agents and web services. In *Proceedings of the 2005 Workshop on Service-Oriented Computing and Agent-Based Engineering (SOCABE'2005)*, Utrecht, The Netherlands, 2005.
7. 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.
8. 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.
9. 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.
10. M. Juric, P. Sarang, and B. Mathew. *Business Process Execution Language for Web Services 2nd Edition*. Packt Publishing, 2006.
11. 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.
12. V. Mascardi and G. Casella. Intelligent agents that reason about web services: a logic programming approach. In *Proceedings of the ICLP'06 Workshop Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services (ALPSWS2006)*, pages 55–70, 2006.
13. S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.
14. 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.
15. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
16. 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.

17. 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.
18. 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.
19. M. B. van Riemsdijk. *Cognitive Agent Programming: A Semantic Approach*. PhD thesis, 2006.
20. 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.
21. 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.
22. M. B. van Riemsdijk and M. Wirsing. Goal-oriented and procedural service orchestration: A formal comparison, 2007. `http://www.pst.ifi.lmu.de/~riemsdijk/goalproc.pdf`.
23. 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.
24. 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.
25. M. Wirsing, A. Clark, S. Gilmore, M. Hölzl, A. Knapp, N. Koch, and A. Schroeder. Semantic-based development of service-oriented systems. In *Formal Techniques for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *LNCS*, pages 24–45. Springer-Verlag, 2006.