Semantic Web Services Challenge 2008

Abductive Synthesis of the Mediator Scenario with jABC and GEM

Christian Kubczak[1] Tiziana Margaria[2] Matthias Kaiser[3] Jens Lemcke[4] Björn Knuth[2]

[1] Chair of Software Engineering, Dortmund Univ. of Technology, 44227 Dortmund, Germany, christian.kubczak@cs.uni-dortmund.de

[2] Chair of Service and Software Engineering, Universität Potsdam, 14482 Potsdam, Germany, margaria@cs.uni-potsdam.de

[3] SAP Res. Center Palo Alto, SAP Labs LCC, 3410 Hillview Ave, Palo Alto, CA 94304, USA, matthias.kaiser@sap.com

[4] SAP Research, CEC Karlsruhe, SAP AG, Vincenz-Prießnitz-Str. 1, 76131 Karlsruhe, Germany, jens.lemcke@sap.com

Abstract. We reuse here the framework, the setting, and the semantic modelling for the automated synthesis of the SWS Challenge Mediator presented in the companion paper [5], and show how to exchange the synthesis paradigm from a linear-time-logic proof based algorithm to a runtime, abductive synthesis implemented in Prolog, that responds directly to situational changes. Both solutions take advantage of a policy-oriented enterprise management approach.

1 Motivation

The Semantic Web Service Challenge Mediation Scenario has been solved in [5] by synthesis of its business logic within jABC [1]. There, we use semantic annotations for data and services expressed via type and activity taxonomies, and we derive automatically several versions of the business logic by means of a proof algorithm for the (semantic) linear time logic SLTL. We showed thus that, given high-level, declarative *knowledge* about the goals to be achieved by the mediator, and abstract information on the available services, their purpose, and their compatibility conditions, we could automatically produce the set of processes that solve the goal and satisfy the constraints. This way, we produced a set of solutions that are correct by construction, are at the same time executable service orchestrations, and are amenable to investigation for compliance wrt. additional criteria, since these orchestrations can be directly analyzed with jABC's mechanisms, e.g. by model checking.

Such an approach has the typical advantages of design-time solutions: they can be saved, compiled for specific platforms, thus living outside the environment where they are computed, and are adequate for documentation, auditing, and massive reusal.

In cases where the environment is highly dynamic it may however be desirable to have a more reactive generation of solutions. Typically this leads to on-the-fly approaches, where the (piecewise) creation of the solution and its (possibly piecewise) execution happen simultaneously. SAP's Goal-oriented Enterprise Management (GEM)

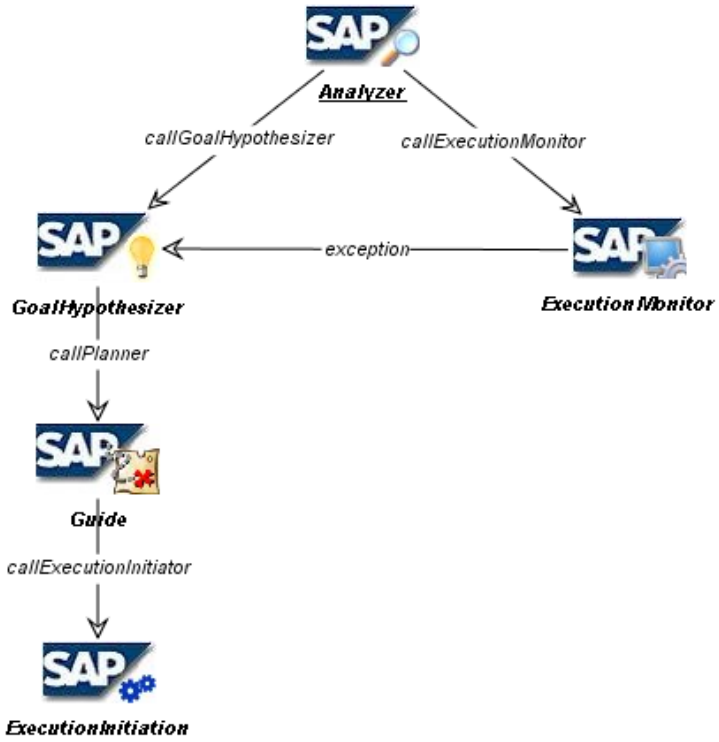

Fig. 1. GEM Orchestration Graph in the jABC: The workflow of a GEM agent

approach, that follows the principles introduced in [2, 3], is a representative of this class. A typical advantage of such approaches is their situation awareness: changes in the environment are directly taken into account while building the solution. The typical drawback is that the existence of a solution cannot be guaranteed, and if the solution space empties, expensive backtracking and rollbacks are needed. Also, justification and further analysis can only be carried out by a-posteriori validation.

An ideal environment should therefore offer both paradigms and enable the user to choose from case to case. This is what we want to achieve with the jABC. To this aim, it is important that both synthesis approaches foot on the same semantic descriptions, use the same information in roughly the same way, and can share the same grounding.

In this paper, we show how both approaches are instances of business knowledge-driven process synthesis [6, 3], and how we can accommodate both in the jABC environment in a modular way: we exchange the synthesis method with SAP's GEM approach to automatically solve the same mediation task using the same domain modelling and service grounding.

1.1 Agent-based Mediator Synthesis.

One first step towards the solution is to implement the idea behind this on-the-fly concept within a working prototype. The basic idea is to construct a set of agents that interact with the company's system and find parts of workflows as solutions to *goals* in a dynamic environment. The concept behind the behavior of an agent is presented in [3] and illustrated here in Fig. 1. In this reference model, a GEM agent refers to situations and goals in order to identify suitable subgoals that, in the given situation, can be reduced via business rules resulting in a fragment of a plan that can be enacted.

We implemented the GEM reference agent as a jABC orchestration: there is a working jABC process, implemented as a service, for each single agent component, shown later in fig. 4, and the jABC orchestration of Fig. 1 corresponds to the workflow of the reference description. Each agent component *Goal Hypothesizer, Guide, Explainer, Analyzer, Performer* is implemented as a Prolog algorithm that is accessed in the jABC as a jETI service hosted on a remote machine. External resources are the collections of BusinessExecutionPolicies, AgentScopePatterns, AgentPurposeRules, that feed the single agents with knowledge, and the collection of BusinessObjects on which the agents operates. They are also hosted elsewhere and accessed as services. Additionally, Explanations that justify the decisions taken can be consulted.

In order to understand how this works in practice, we need first to introduce abductive reasoning, the event calculus, and to explain how the GEM synthesis process is organized.

In the following, Section 2 sketches abductive reasoning and the event calculus, Section 3 describes the Goal-oriented Enterprise Management system (GEM), and Section 4 shows how to use GEM to solve the first Mediation scenario. Finally a conclusion is drawn and the ongoing tasks are presented in section 5.

2 Abductive reasoning

In contrast to the traditional composition, Our general approach towards synthesizing business processes is based on the notion of abductive reasoning. We synthesize a goal-driven process on the basis of available services represented as descriptions of pre- and postconditions (or conditions and effects), as well as policies which constrain which goals are admissible (main goal and subgoals) and how to achieve them. We now take a brief look at the nature of abductive reasoning, how it is realized in ways of programming and then describe the process synthesis by ways of a planner implementing a form of abductive process synthesis.

Abductive reasoning is also called *reasoning to the best explanation*. It is the kind of reasoning used to hypothesize causes on the basis of observed results. Thus one of its main applications is in diagnosis, where symptoms are observed and their causes are abduced = hypothesized. A simple way to illustrate abductive reasoning in a more logical way is to imagine the abduction step as *modus ponens* in a backward manner. In modus ponens, given $P -> q$, once we observe P, we can deduce Q. In abduction, given $P -> q$, once we observe Q we infer that P is an admissible cause of Q.

The relation between deduction, abduction, and induction (another form of reasoning forming rules from observations) has been studied extensively by Peirce. The only sound kind of reasoning is deduction. However, the only way to obtain new knowledge is by 1) using abduction and induction and 2) then prove the validity of newly obtained facts or rules within a previously formed theory (where refinement of this theory is also an option in case of conflicts with new facts/rules).

A whole area in logic programming concerns the use of abduction as form of reasoning to enhance given theories by new, yet not integrated information. For the synthesis of business processes we employ the general idea of abductive logic programming.

2.1 Abductive logic programming.

Abductive logic programming is an extension of logic programming which is characterized by the possibility that predicates which are not completely defined can be used to explain other predicates. I.e. the explaining (or abducible) predicates are derived by abduction. Abductive logic programs typically consist of:

- a logical program, consisting of facts and rules which can be used in the common deductive way,
- a set of abducible predicates which are incompletely defined (think of them as *if* parts in *if-then* rules)
- a set of integrity constraints which determine certain properties the program must comply with.

We now apply these notions in our domain of business process synthesis to see the connection.

We have a description of the world in form of an ordinary logic program. Situations are described by facts holding at a certain time. Services can be seen like (very complex) if-then rules with the preconditions being abducible predicates. Likewise, goals can be seen as abducible predicates which may explain other goals. Policies can be seen as constraints whose violations would lead to non-admissible programs.

Important to know is that in order to achieve a goal, i.e., complete the abducible predicate, a goal often has to be decomposed into a number of subgoals which can be achieved - normally in a certain order - so that one subgoal can be explained on the basis of the achievement of another.

If we have a business goal that we want to be achieved with a business process, then what we expect is a process that achieves the goal by execution of a sequence of services that transform an initial situation into a situation in which the goal is consistent. Such a synthesis of a process can be realized by a planning algorithm.

The algorithm underlying our planning strategy is derived from the well-known *event calculus*, proposed in [4] and extended by many for various purposes.

2.2 Event Calculus in Short.

Generally, the event calculus facilitates the representation and reasoning about dynamic entities involving actions and their effects using (a subset of) first order logic. This

makes it a promising approach for planning. The language of the event calculus allows for the representation of time independently of any of events occurring.

The most common representation of event calculus can be briefly described as follows:

1. a set of *time points* which can be denoted by integers,
2. a set of *properties*, called fluents, which can vary over time, i.e., from one time point to another, and
3. a set of event types.

The event calculus is based on the notion of fluents which can be manipulated by actions in time. Fluents are usually functions which are specified by the predicate *holds* to be in effect at a certain time point. Fluents can be changed by actions. A fluent is initiated, i.e., made to hold by means of an *initialize*, and can be terminated by a *terminate* action. If we 'count' actions at time steps we can say whether a certain fluent is true or false at a given time point. Translated into our domain this means that we can regard services as actions, while looking at preconditions and effects as fluents. The event calculus takes a number of domain-independent axioms as a foundational theory, which is a logic program. It is enriched by domain-dependent information about entities like constituents of situations, service preconditions and effects, and service descriptions that state which fluents hold before and after the execution of a service.
Additionally, constraints in terms of policies can be added to "control" which changes can legitimately happen at certain time points.

It is now straightforward to realize a planner over goals, services with their preconditions and effects in compliance with formalized policies.

The planner starts the plan generation when a goal is supplied. It tries to find a service which initiates those fluents that are described in the goal. If such a service is found, it will be checked whether the preconditions of that service, which are expressed as fluents, hold. If not, another service must be found whose execution will ensure that the fluents hold and so forth. The whole planning process can be much more complex, since more than one service might be required to satisfy goals and subgoals by initiating required fluents. Note that ordering of services is ensured through time points that are attached to service invocations.

The plan is complete if preconditions for a service are actually contained in the description of the initial situation. This way, trough abduction we arrive at a plan that can now be executed to transform the initial situation into a situation in which the supplied goal can be achieved. The whole plan is represented in a declarative form and it can be proven to be correct given the resources utilized because, while the synthesis was based on abduction, the proof can be done in a deductive way.

In the concrete SWS Mediation Scenario, a final goal for the SWS Challenge Mediation is to transform a *PIP3A4PurchaseOrderRequest* into a *PIP3A4PurchaseOrderConfirmation*. As this is a rather complex thing to do for a single on-the-fly agent (planner), this goal is split up into several subgoals that are solved individually. Collectively, they yield the desired solution.

3 The GEM approach

3.1 Situation Description.

The prerequisite for establishing a GEM system is that an enterprise's current data, called "situation", and its IT systems' capabilities, e.g., Web service operations, are specified using the event calculus, e.g.

```
Situation: [por(por1)] Operation: [createOrder(POR),
            (requires,[por(POR)]),
            (terminates,[]),
            (initializes,[po(PO),po_basedOn(PO,POR)])]
```

3.2 GEM Agents.

A GEM agent is a generic software with a memory that observes the enterprise's situation, and influences it through operation invocations. An agent is configured with respect to the types of relevant observations, the business goals to pursue upon an observation, and the business execution policies to maintain when influencing the situation.

The processing of a GEM agent is structured into the five components Analyzer, Goal Hypothesizer, Guide, Performer (comprising the Execution Initiator, and the Execution Monitor), Explainer depicted in Fig. 1. The components are realized as PROLOG programs. They store their internal results in the memory of the agent. Concretely,

- The *Analyzer* probes the situation based on the configured relevant observation, e.g., reacting on a purchase order request `?-por(POR)`, and stores the observations, e.g., `[por(por1)]`.
- The *Goal Hypothesizer* generates a business goal based on both the configuration and the observation, e.g., the goal is to trigger the internal purchase order management denoted by a purchase order document in the situation, and the confirmation of the purchase order request denoted by a purchase order request document in the situation `[po(po1),poc(poc1)]`.
- The Guide invokes an *event calculus planner* [7] with the inputs of the IT system's capabilities, the situation, the business execution policies, and the goal, resulting in a partial-order plan, e.g.,

  ```
  [[happens(createOrder(por1),t1),
    happens(sendFeedback(por1),t2)], [before(t1,t2)]]
  ```

- The *Execution Initiator* creates a process execution instance in the memory of the agent based on a plan, e.g., stating that the execution currently stands before the first operation's time `[(before,t1)]`. In addition, the Execution Initiator tells the Analyzer to wait for the precondition of the first operation in the plan to occur.
- The *Execution Monitor* only reacts on observations made by the Analyzer that were requested by the Execution Initiator. This is in contrast to the Goal Hypothesizer, which reacts only to observations made due to the agent's scope configuration.

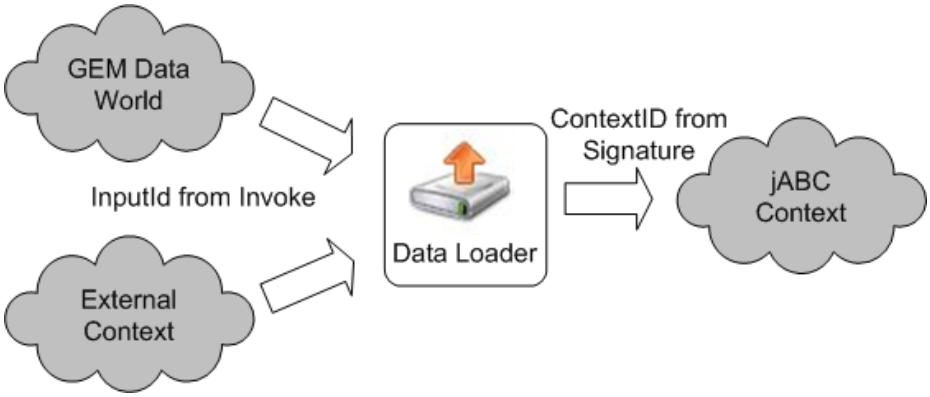

Fig. 2. Loading the data into jABC

The Execution Monitor's processing depends further on the observation made. If a precondition was observed, the respective operation is invoked in the IT system, e.g., by a Web service call, and the process execution instance is updated, e.g., `[(after,t1)]`. In this case, the Analyzer is asked to wait for the postcondition of the just invoked operation. If a postcondition was observed, the process execution instance is updated, e.g., `[(done,t1),(before,t2)]`, and the Analyzer is asked to wait for the postcondition of the respective operation.

Partial order plans are correctly tracked by triggering multiple preconditions when the postcondition of an operation preceding a fork is observed, and only invoking an operation following a join when all operations preceding the join are known to be finished.

3.3 Hiding the Agent Structure.

As shown in Fig. 4, several GEM agents can be hiearchically encapsulated inside the jABC within a single *GraphSIB* each, and thus their complexity and internal organization can be hidden from the end users. The overall process becomes a black box, and could be hosted itself as a service on some remote machine.

4 Using GEM to solve the Mediation

This generic GEM Agent provides a generic on-the-fly synthesis capability. Its implementation is completely independent from the concrete setting and problem to be solved. It behaviour is in fact controlled (and this way parameterized) by the knowledge basis, in particular by the external policies that describe the enterprise system under consideration.

We use GEM's *Prolog* reasoner to 1) derive specific subgoals and 2) call Moon's web services. This is a problem-oriented decomposition, since the specification of the

Web service semantics, the enterprise's goals and policies happens on a higher level not mentioning *concrete operations* and their permissible sequences, but rather talking about desired and unwanted dependencies of the *business objects* manipulated through the operations using the event calculus (see below), that takes here the role played by SLTL in the companion approach.

As such, the entire GEM approach is embedded within the jABC, and the *synthesis process* is modeled in the jABC.

4.1 Integrating the GEM system in jABC.

To use the Prolog implementation of GEM in the Java based jABC, a wrapper class encapsulates the calls of any GEM agent procedure: Situation Analyzer, Goal Hypothesizer, Guide, Execution Initiator and Execution Monitor. This is the same technique we used in [5] to call the LTL synthesis algorithm, which is implemented in C++ and runs on a remote machine.

Each agent procedure is embedded in an own SIB in the jABC, so that this way we have a GEM SIB-palette in the jABC and we can call every agent procedure individually. The orchestration of the GEM agent procedures is modelled in the jABC as a service logic graph: its workflow is presented in Fig. 1. The GEM system is executed on a remote machine and called via web services, so GEM users need not take care of any Prolog installation: this is virtualized by the jABC

4.2 The GEM Synthesis Process in jABC.

In every agent execution, the Analyzer is the first procedure to be executed (thus the start SIB, underlined): here the situation is observed. As explained in Sect. 3, the Analyzer decides that a new plan has to be created by calling the Goal Hypothesizer or to follow the plan execution by calling the Execution Monitor. Alternatively the Analyzer's observation can end up with the result that nothing is done for the agent. So the three SIBs Analyzer, Execution Initiator, and Execution Monitor are legal final SIBs for the GEM orchestration graph. Additionally, every SIB can terminate the workflow in case of an error. In case of the observation of an exceptional pattern, first the Execution Monitor is called to reset the execution to the last goal, and then the Goal Hypothesizer is called to create a new plan.

4.3 Service invocation: The invocation of a (Moon) Operation.

When the Execution Monitor requests an (abstract) operation, a concrete service must be called. Solving the problem in general turns out to be more complex than first expected. Besides the communication of the invocation request itself (service invocation), jABC has to manage storing the data and mapping between the data representation of GEM and the real world (service binding and communication).

To communicate an invocation request, the SIB wrapper offers an own Java class, Invoke, containing the name of the operation, a list of input identifiers, a list of output identifiers and finally an operation identifier. The operation identifier is used by

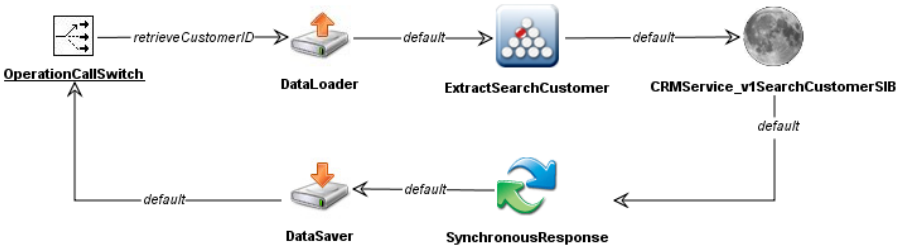

Fig. 3. Exemplary Moon invocation sequence

jABC to call the synchronous response method, also offered by the interface. Using this method, jABC informs GEM about the success of the operation.

As we can see, GEM and jABC only exchange *identifiers* for the real world objects, like input and output values. The advantage is that GEM does not need to store complex objects nor send complex messages. So the planner only has to include and manage the (abstract, minimal) information needed for decision making. This decouples GEM from concrete data issues.

On the other hand we need to map the GEM id's to the real world data, and of course store the data and make it accessible during execution. This is usually done via the internal *execution context* of the jABC. However, to enable other (external) systems to enhance the situation for GEM with further knowledge and to store it, we introduce a similar *external execution context*, which is accessible via web services. As shown in Fig. 2, for every requested data, like SearchCustomerResponse or LineItem, an own getter and setter service exists, to load and save the data using the id's assigned by GEM with the Invoke-object.

Two generic DataLoad SIBs were created, respectively to Load the input data into the jABC context of the GEM agent execution, and to Save it from the jABC to the external context. These SIBs must know which context-ids to use when accessing the jABC context. To this aim, the signatures of all operations on which the GEM operations and the Operation-SIBs in jABC are based upon are initially imported, and their parameter names are then taken to load/save the objects from/to the jABC context.

One more SIB is necessary to guarantee a completely generic GEM agent workflow: the Operation Call Switch. This SIB simply forwards the invocation to the right Service operation, on the basis of a description of the Operations supported by an external agent. Fig. 3 shows the exemplary invocation request of the Moon Service Search Customer, together with the operation selection Operation Call Switch and the Load/Save SIBs.

Once an operation is invoked, the Operation Call Switch is called again: since GEM may work with multiple threads, the Execution Monitor could send more than one invocation request in one message. The Operation Call Switch then decides to call the next operation or, if this was the last operation, return control to the next agent.

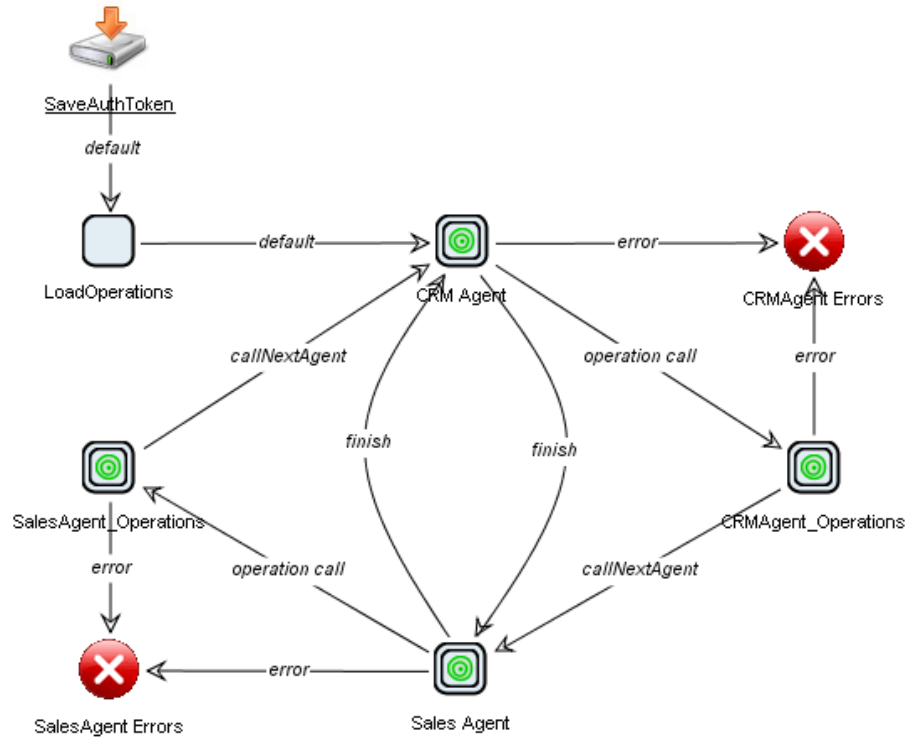

Fig. 4. Mediation Scenario 1 with GEM/jABC

4.4 GEM at work for the Mediator.

To remain close to the Mediation scenario description, two Moon agents are created:

- a *CRM agent* that communicates with Moon's customer relationship management system, and
- a *Sales agent* that communicates with Moon's Order Management System.

Since jABC supports hierarchical Service Logic Graphs, we hierarchically encapsulate the entire GEM agent graph in a single GraphSIB: this way both agents use the same model of Fig. 1, which is instantiated twice. A model parameter determines the type of the agent.

As we see in Fig. 4, when a Blue Purchase order request arrives, the SIB SaveAuthToken starts the planning of the mediator solution. We have two instances of the GEM agent of Fig. 1, one for the CRM Agent and one for the Sales Agent, each of them determining

- either the next operation to be executed by that agent, passing the control to the own operation executor (SIBs CRM Agent Operations and Sales Agent Operations)

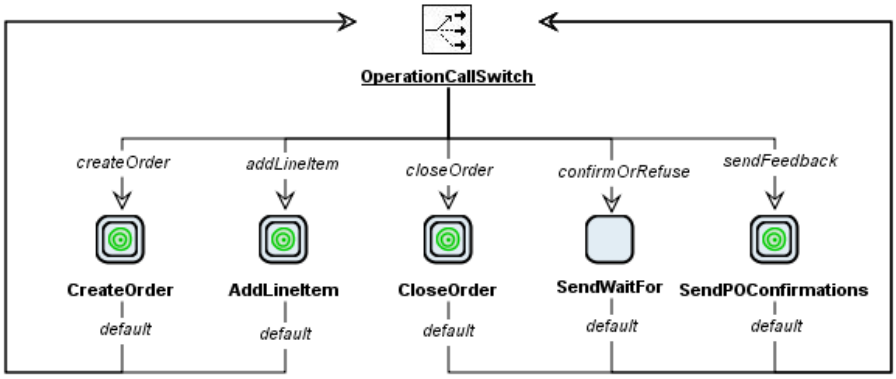

Fig. 5. Operations of the Moon sales agent (i.e., its SIB Library

– or the own termination, passing then the token to the next agent.

During execution, one agent is thus able to complete one step of its execution, then passing the execution to the next agent. This rotation prevents agents from taking a lot time completing multiple tasks, while other agents wait. In fact, at any time the situation could be enriched by another participant, giving an agent new tasks and inducing longer delays for the other agents, up to starvation. The scheduling strategy can however be easily changed by modifying this graph.

The set of operations of an agent is itself modelled within a GraphSIB, see Fig. 4. While Fig. 3 shows the only operation (SearchCustomer) of the CRM agent, the Sales agent has much more to do. As shown in Fig. 5 the sales agent has to call the four web service operations CreateOrder, AddLineItem, CloseOrder, and SendPOConfirmation. Additionally the SIB SendWaitFor is called to be able to react on a asynchronous response, here used by the service which receives the confirmation of any line item.

Each of these operations is implemented along the lines shown in Fig. 3: it is in fact an invocation pattern comprising a Load-SIB, potential SIBs for data mediation, the specific Operation-SIB, a Save-SIB and a SIB for the synchronous response.

4.5 Mediator Execution: Communication with the testbed.

To communicate with the SWS Challenge testbed, two more web services are created:

– The first receives the PurchaseOrderRequest message of the Blue Company, writes the relevant data into the GEM situation and stores the corresponding objects into the external context. This is the only place where the situation is written outside the GEM system.
– The second web service receives the confirmation of any line item and informs GEM about it. The receipt of a line item confirmation is designed using a procedure for an asynchronous response of the AddLineItem operation. The interface

offers a further method for that purpose and an additionally class WaitFor (similar to invoke), with which GEM informs jABC that the agent waits for a specific operation to be done. So every operation can have a synchronous and asynchronous response.

5 Conclusion

We showed here how to reuse the framework, the setting, and the semantic modelling for the automated synthesis of the SWS Challenge Mediator presented in [5], in order to exchange the synthesis paradigm from a linear-time-logic proof based algorithm to an abductive synthesis implemented in Prolog, that responds directly to situational changes.

Both synthesis approaches have the potential to become a highly flexible middle layer for large scaling business processes, combining the advantages of agent technology and service-oriented computing. By enhancing the underlying reasoning technology the prototype has the chance to become a declarative mediator controllable by non-programmers, like business experts. Since it is situation aware, it can react promptly to situational changes, and this way potentially self-adapt during execution.

The implementation is still prototypical, and restricted so far to Mediation scenario Part 1. We are currently working on extending it to the other parts of the scenario, and on a hybrid approach that takes advantage of the dynamic agent behavior within a full service-oriented framework that considers even the reasoning algorithms as retargetable services.

References

1. Sven Jörges, Christian Kubczak, Ralf Nagel, Tiziana Margaria, and Bernhard Steffen. Model-driven development with the jABC. In *HVC - IBM Haifa Verification Conference*, LNCS, Haifa, Israel, October 23-26 2006. IBM, Springer Verlag.
2. M. Kaiser and J. Lemcke. Towards a framework for policy-oriented enterprise management. In *AAAI*, 2007.
3. Matthias Kaiser. Towards the realization of policy-oriented enterprise management. *IEEE Computer Society Special Issue on Service-Oriented Architecture forthcoming*, 11:65–71, 2007.
4. Robert A. Kowalski and Marek J. Sergot. A logic-based calculus of events. *New Generation Comput.*, 4(1):67–95, 1986.
5. T. Margaria, M. Bakera, H. Raffelt, and B. Steffen. Synthesizing the mediator with jabc/abc. In *EON-SWS Workshop Semantic Web Service Challenge Workshop, this volume*, June 2008.
6. T. Margaria and B. Steffen. Service engineering: Linking business and it. *IEEE Computer, issue for the 60th anniversary of the Computer Society*, pages 53–63, October 2006.
7. Murray Shanahan. Event calculus planning revisited. In Sam Steel and Rachid Alami, editors, *ECP*, volume 1348 of *Lecture Notes in Computer Science*, pages 390–402. Springer, 1997.