Synthesizing Semantic Web Service Compositions with jMosel and Golog

Tiziana Margaria[1], Daniel Meyer[1], Christian Kubczak[2], Malte Isberner[2], and Bernhard Steffen[2]

[1] Chair Service and Software Engineering, Universität Potsdam, Germany
{margaria,meyerd}@cs.uni-potsdam.de
[2] Chair of Programming Systems, TU Dortmund, Germany
{christian.kubczak,malte.isberner,steffen}@cs.tu-dortmund.de

Abstract. In this paper we investigate different technologies to attack the automatic solution of orchestration problems based on synthesis from declarative specifications, a semantically enriched description of the services, and a collection of services available on a testbed. In addition to our previously presented tableaux-based synthesis technology, we consider two structurally rather different approaches here: using *jMosel*, our tool for *Monadic Second-Order Logic on Strings* and the high-level programming language *Golog*, that internally makes use of planning techniques. As a common case study we consider the Mediation Scenario of the Semantic Web Service Challenge, which is a benchmark for process orchestration. All three synthesis solutions have been embedded in the jABC/jETI modeling framework, and used to synthesize the abstract mediator processes as well as their concrete, running (Web) service counterpart. Using the jABC as a common frame helps highlighting the essential differences and similarities. It turns out, at least at the level of complication of the considered case study, all approaches behave quite similarly, both considering the performance as well as the modeling. We believe that turning the jABC framework into experimentation platform along the lines presented here, will help understanding the application profiles of the individual synthesis solutions and technologies, answering questing like when the overhead to achieve compositionality pays of and where (heuristic) search is the technology of choice.

1 Introduction

Dealing with (Web) services, semantics is gaining terrain as technology for application development and integration. However, semantics-based technology is still highly complicated in usage and implementation, which explains its relatively slow industrial adoption. It is the goal of the Semantic Web Service Challenge (SWSC) to overcome this situation by studying concrete case studies, with the goal of pinpointing the application profiles of the various approaches proposed so far. The corresponding leading case study is the Mediation Scenario [1], a business orchestration problem that requires adequate "translation" between different communication protocols and data models.

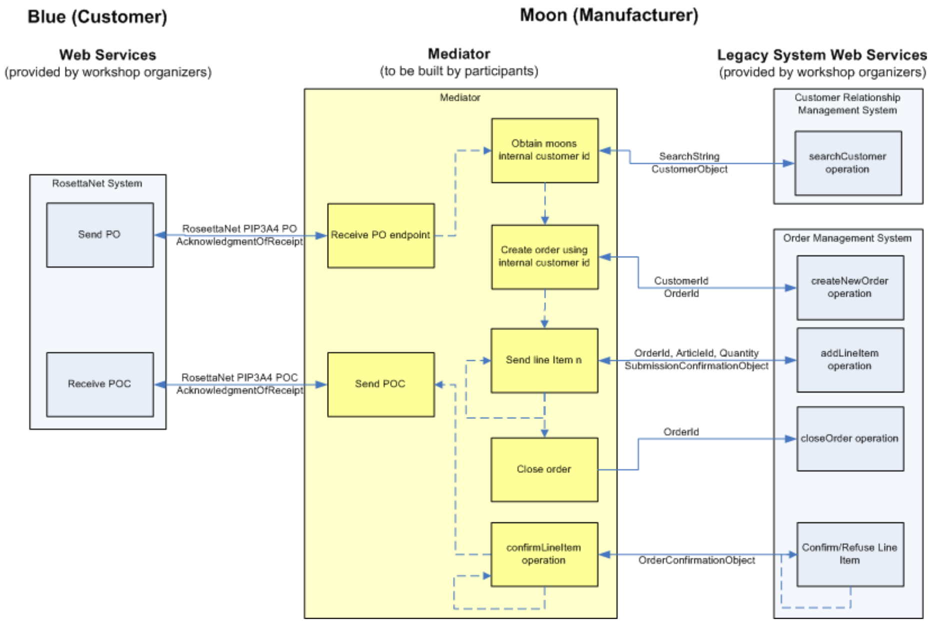


Fig. 1. The Mediation Scenario of the SWS Challenge

In this scenario, a customer named *Blue* (Fig. 1(left)) submits business orders in format and protocol compliant to the RosettaNet Pip3A4 protocol [2] to the backend system of*Moon* (Fig. 1(right)). Since *Moon* does not support RosettaNet the Pip3A4 communication standard, the task of the challenge is to automatically generate a mediation service that enables *Moon*'s legacy system to process the requests submitted by *Blue*.

This problem has been addressed by several research groups in the past: 5 solutions, 2 of which ours, have been presented and compared in [3]. We first directly modeled the mediator's orchestration in the jABC [4,5,6,7], our framework for service oriented development and model-driven orchestration, and generated and published the mediator using our jETI technology. In jABC, processes are orchestrations, they are modeled as directed flow graphs called Service Logic Graphs (SLG), where the nodes, which represent the services, are called *Service Independent Building Blocks* (SIBs).

Second, we enhanced the capabilities of the jABC by integrating a synthesis algorithm based on Semantic Linear Time Logic (SLTL) specifications [8]. Using this synthesis method, we were indeed able to automatically (re-)produce an equivalent orchestration [9]. This observation got us interested in using also other very different process/model synthesis techniques on the same problem, in order to compare (following the original purpose of the SWS Challenge) different methods and techniques, based on different semantic representations of declarative knowledge and goals.

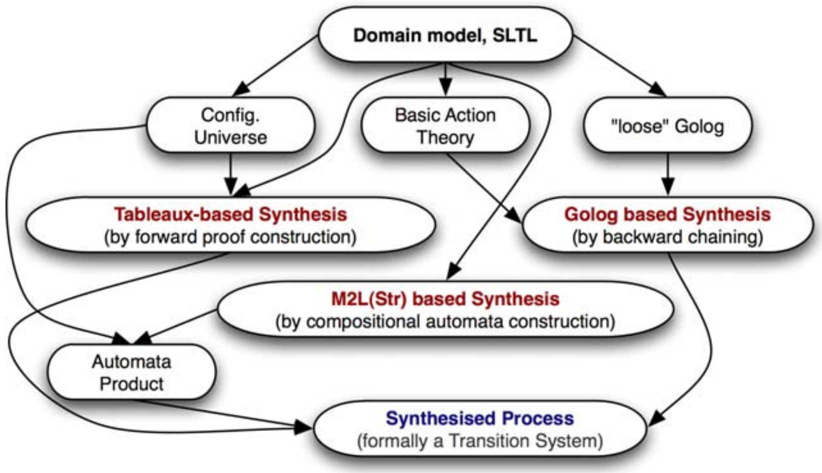

Domain model, SLTL
Config. Universe
Basic Action Theory
"loose" Golog
Tableaux-based Synthesis (by forward proof construction)
Golog based Synthesis (by backward chaining)
M2L(Str) based Synthesis (by compositional automata construction)
Automata Product
Synthesised Process (formally a Transition System)

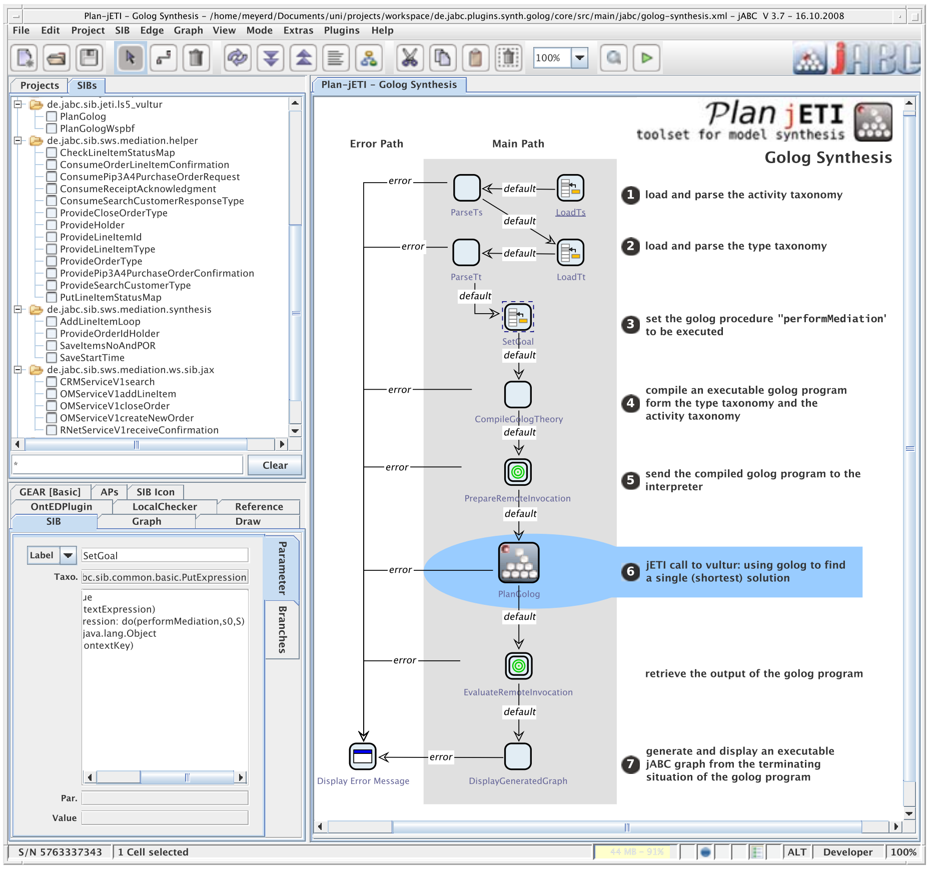
Fig. 2. The landscape of synthesis techniques in jABC

In this paper, we apply two alternative and conceptually very different approaches to planning to the Mediation Scenario:

– an approach based on monadic second order logic on strings M2L(Str) [10], which works by compositional automata construction in jMosel [11], and
– an approach based on Golog [12], a tool that internally uses backward chaining for solving planning/synthesis tasks. This work is based on the well known situation calculus planner by Reiter.

Fig. 2 provides a conceptual sketch covering the essence the three approaches we considered in details so far. Our study revealed that despite the huge conceptual difference these three approaches share quite some similarities. In particular, they were all able to quite naturally express the original problem in sufficient detail to automatically synthesize the desired solution to the Mediation problem (cf. Fig. 6). However there are also similarities at a different level, which became apparent when modeling the Golog-synthesis process itself as a process using jABC. To this aim, we implemented the steps described so far as a set of independent, reusable building blocks (SIBs). The resulting 7-step synthesis process is shown in Fig. 3. It is the same process we obtained with jMosel and with the previous (S)LTL based synthesis methods. Thus this process can be considered as a pattern for synthesis solutions, which allows one to integrate and/or combine various synthesis and transformation functionalities like the ones summarized in Fig. 2 to complex heterogeneous solutions. These solutions are then directly executable inside the jABC, or, as we did in the previous phases of the Challenge, they can be exported as a Web service.

After presenting our modeling framework jABC in Section 2, we show how we can easily plug in different algorithms and knowledge representation formalisms, namely *jMosel*, see Section 3 and *Golog*, Section 4. Subsequently we show how


Plan-jETI – Golog Synthesis – /home/meyerd/Documents/uni/projects/workspace/de.jabc.plugins.synth.golog/core/src/main/jabc/golog-synthesis.xml – jABC V 3.7 – 16.10.2008
File Edit Project SIB Edge Graph View Mode Extras Plugins Help
100%
jABC
Projects SIBs
de.jabc.sib.jeti.ls5_vultur
PlanGolog
PlanGologWspbf
de.jabc.sib.sws.mediation.helper
CheckLineItemStatusMap
ConsumeOrderLineItemConfirmation
ConsumePip3A4PurchaseOrderRequest
ConsumeReceiptAcknowledgment
ConsumeSearchCustomerResponseType
ProvideCloseOrderType
ProvideHolder
ProvideLineItemId
ProvideLineItemType
ProvideOrderType
ProvidePip3A4PurchaseOrderConfirmation
ProvideSearchCustomerType
PutLineItemStatusMap
de.jabc.sib.sws.mediation.synthesis
AddLineItemLoop
ProvideOrderIdHolder
SaveItemsNoAndPOR
SaveStartTime
de.jabc.sib.sws.mediation.ws.sib.jax
CRMServiceV1search
OMServiceV1addLineItem
OMServiceV1closeOrder
OMServiceV1createNewOrder
RNetServiceV1receiveConfirmation
Clear
GEAR [Basic] APs SIB Icon
OntEDPlugin LocalChecker Reference
SIB Graph Draw
Parameter Branches
Plan-jETI – Golog Synthesis
Plan jETI
toolset for model synthesis
Golog Synthesis
Error Path
Main Path
error
default
ParseTs
LoadTs
ParseTt
LoadTt
SetGoal
CompileGologTheory
PrepareRemoteInvocation
PlanGolog
EvaluateRemoteInvocation
Display Error Message
DisplayGeneratedGraph
S/N 5763337343
1 Cell selected
ALT Developer 100%

Label SetGoal
Taxo. bc.sib.common.basic.PutExpression
ue
textExpression)
ression: do(performMediation,s0,S)
java.lang.Object
ontextKey)
Par.
Value

1 load and parse the activity taxonomy

2 load and parse the type taxonomy

3 set the golog procedure "performMediation" to be executed

4 compile an executable golog program form the type taxonomy and the activity taxonomy

5 send the compiled golog program to the interpreter

6 jETI call to vultur: using golog to find a single (shortest) solution

retrieve the output of the golog program

7 generate and display an executable jABC graph from the terminating situation of the golog program

Fig. 3. The Golog-based synthesis process modeled in jABC

easily these at first sight completely different technologies can be operated on top of the same platform and can be used to obtain the same results, see Sect. 5.

2 Our Modeling Framework

Basic ingredient for the process mediation, e.g. for bridging the gap between Blue and Moon, are:

– a set of business objects, including the definition of their structure, properties, and data types,
– a set of services (SIBs) operating on the business objects, including knowledge about relevant properties, and
– domain knowledge, concerning semantic properties of the domain under consideration (typically, information on the data and on how the data can be manipulated by the services), and behavioral knowledge (often also called

procedural knowledge), that describes abstractly properties (restrictions, precedences, lose constraints) of the collaboration of such services.

Behavioral knowledge is often present in a domain description, and often the task to be performed by the orchestration is also already known in its behavioral traits. From the point of view of a user of semantic techniques, all the available knowledge should be optimally exploited by the techniques of a semantic web framework, in order to ensure optimal precision and efficiency. Thus, procedural knowledge is in our opinion much more than a way of formulating domain specific heuristics to make process synthesis computationally feasible: in real world applications, we need to be able to formulate constraints which must be satisfied by processes in order to be admissible. For example we may need to ensure

- general ordering properties (e.g., ensuring that some service is executed before another)
- abstract liveness properties (e.g., guaranteeing that a certain service is eventually executed)
- abstract safety properties (e.g., making sure that certain services are never executed simultaneously).

However, we do not want to specify the whole processes in a static, fixed way. On the contrary, we use *loose coordination specifications*, which leave room for variability inside the bounds of a declarative specification that includes behavioral constraints.

In jABC, we have implemented this adopting as semantic predicates abstraction concepts from *dataflow analysis* [13] in a *constructive* way, leading to the domain knowledge representation introduced in [8,14,15] and summarized in Sect. 2.1.

2.1 Modeling Domain Knowledge with Taxonomies

In jABC's modeling style, *business objects* are mapped to abstract semantic concepts that we call *Types*, and services, actually the collection of SIBs corresponding to single service operations, are mapped to semantic *activities*. Both types and activities are organized in *taxonomies*.

A taxonomy $\mathcal{T}ax = (T, CT, \rightarrow)$ is a directed acyclic graph (DAG) where CT is a set of concrete entities that are grounded to instances of the real world (as the elements of the A-box of Description Logics), and that are the sinks in the graph, T is a set of abstract concepts, and $\rightarrow$ relates concepts and concrete elements (T, CT) or pairs of concepts (T, T).

In a type taxonomy, as shown in Fig. 4 for the mediation problem, $\mathcal{T}_{ty} = (T_t, CT_t, is\_a)$, where CT_t is a set of semantic types that directly correspond to individual business objects, T_t is a set of abstract semantic types that represent groups of business objects, and edges reflect an *is_a* relationship. In our example, *OrderIDs* and *Confirmations* are in the group *Orders*. The concrete types

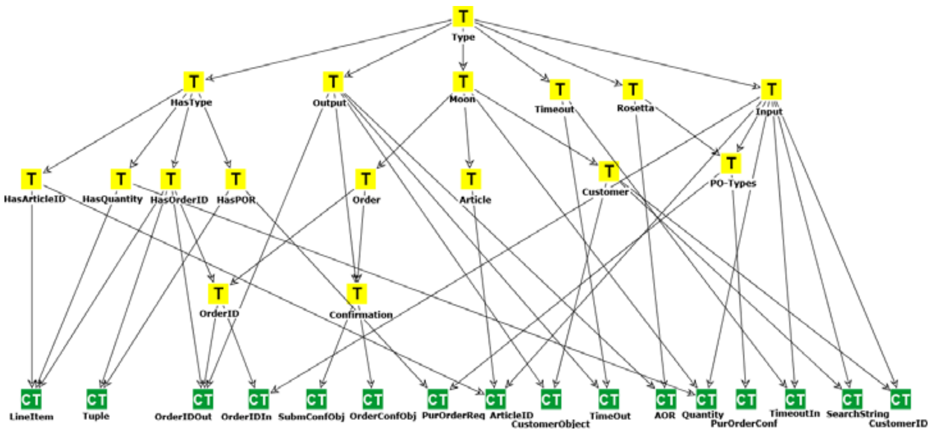


Fig. 4. The Type Taxonomy for the mediation scenario

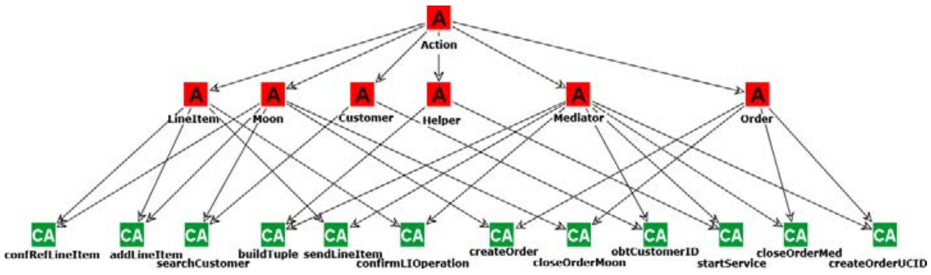


Fig. 5. The Service Taxonomy for the mediation scenario

(the leaves of the taxonomy)are the most concrete semantic entities we want to use in the synthesis problem - here directly the business objects used as input and output by the services.

The activity taxonomy $\mathcal{T}_a = (A, CA, is\_a)$ is defined in the same way, as shown Fig. 5: CA is a set of concrete semantic activities, representing individual SIBs, and abstract activities A represent groups of SIBs which share a common set of (non-)functional properties.

2.2 Dataflow Facts as Semantic Preconditions/Effects

We now need to formulate knowledge about how the activities operate on the types. At a technical level, Web services have very complex relations to business objects. At the semantic level, we abstract these complex relations into three basic functions, well known from dataflow analysis expressing the preconditions and effects of services, which are technically stored in the jABC (semantic) context:

Table 1. The SWS mediation Modules

name	input type (uses)	output type (gen)	description
`Mediator`			Maps RosettaNet messages to the backend
`startService`	$\{true\}$	*PurOrderReq*	Receives a purchase order request message
`obtCustomerID`	*PurOrderReq*	*SearchString*	Obtains a customer search string from the req. message
`createOrderUCID`	*CustomerObject*	*CustomerID*	Gets the customer id out of the customer object
`buildTuple`	*OrderID*	*Tuple*	Builds a tuple from the orderID and the POR
`sendLineItem`	*Tuple*	*LineItem*	Gets a LineItem incl. orderID, articleID and quantity
`closeOrderMed`	*SubmConfObj*	*OrderID*	Closes an order on the mediator side
`confirmLIOperation`	*OrderConfObj*	*PurOrderCon*	Receives a conf. or ref. of a LineItem and sends a conf.
`Moon`			The backend system
`searchCustomer`	*SearchString*	*CustomerObject*	Gets a customer object from the backend database
`createOrder`	*CustomerID*	*OrderID*	Creates an order
`addLineItem`	*LineItem*	*SubmConfObj*	Submits a line item to the backend database
`closeOrderMoon`	*OrderID*	*TimeoutOut*	Closes an order on the backend side
`confRefLineItem`	*Timeout*	*orderConfObj*	Sends a conf. or ref. of a prev. subm. LineItem

– $use(\cdot) : CA \rightarrow \mathcal{P}(CT)$
  in order for the activity a to be executable, values of the set of type $use(a)$ must be present in the execution context
– $gen(\cdot) : CA \rightarrow \mathcal{P}(CT)$, $gen(a)$ returns the set of types, values of which are added to the context after invocation of the activity a
– $kill(\cdot) : CA \rightarrow \mathcal{P}(CT)$, $kill(a)$ is the set of types, values of which are removed form the context if a is invoked

Table 1 lists in the first column a selection of activities from the mediation scenario and maps each activity to a set of input/use types (column 2) and a set of output/gen types (column 3). In this example there are no kill types.

The jMosel approach presented in Section 3 can directly work with this representation of the domain knowledge. In Sect. 4, we show how to translate it into situation calculus and Golog.

2.3 Expressing Behavioral Knowledge

The loose coordination specification language we use to express procedural knowledge is *Semantic Linear Time Logic (SLTL)* [8], a temporal (modal) logic that includes the taxonomic specifications of types and activities.

Definition 1 (SLTL)

The syntax of Semantic Linear Time Logic (SLTL) is given in BNF format by:

$$\Phi ::= \ tt \mid \mathsf{type}(t_c) \mid \neg\Phi \mid (\Phi \wedge \Phi) \mid \langle a_c \rangle\, \Phi \mid \ \mathbf{G}(\Phi) \mid (\Phi\, \mathbf{U}\, \Phi)$$

where t_c and a_c represent type and activity constraints, respectively, formulated as taxonomy expressions.

Taxonomy expressions are for this example propositional formulas that use as propositions the concepts in the taxonomies of Figs. 4 and 5 and the *use*, *gen*, *kill* predicates already introduced.

SLTL formulas are interpreted over the set of all *legal orchestrations*, which are coordination sequences, i.e. alternating type correct sequences of types and activities[1], which start and end with types. The semantics of SLTL formulas can now be intuitively defined as follows[2]:

- $\mathsf{type}(t_c)$ is satisfied by every coordination sequence whose first element (a type) satisfies the type constraint t_c.
- Negation $\neg$ and conjunction $\wedge$ are interpreted in the usual fashion.
- Next-time operator <> :
  $\langle a_c \rangle\, \Phi$ is satisfied by coordination sequences whose second element (the first activity) satisfies a_c and whose *suffix*[3] satisfies Φ. In particular, $\langle tt \rangle\, \Phi$ is satisfied by every coordination sequence whose suffix satisfies Φ.
- Generally operator $\mathbf{G}$:
  $\mathbf{G}(\Phi)$ requires that Φ is satisfied for every suffix satisfies Φ.
- Until operator $\mathbf{U}$:
  $(\Phi\, \mathbf{U}\, \Psi)$ expresses that the property Φ holds at all type elements of the sequence, until a position is reached where the corresponding suffix satisfies the property Ψ. Note that $\Phi\, \mathbf{U}\, \Psi$ guarantees that the property Ψ holds eventually (strong until).
  The frequently occurring formula $(true\, \mathbf{U}\, \Psi)$ is called Eventually and is written $\mathbf{F}\, \Psi$.

The above definition of suffix may seem complicated at first. However, thinking in terms of path representations clarifies the situation: a sub path always starts with a node (type) again. However, users should not worry about these details: they may simply think in terms of pure activity compositions and should not care about the types (which are matched correctly by the synthesis algorithm), unless they explicitly want to specify type constraints.

[1] During the description of the semantics, types and activities will be called *elements* of the orchestration sequence.

[2] A formal definition of the semantics can be found online.

[3] According to the difference between activity and type components, a suffix of a coordination sequence is any subsequence which arises from deleting the first 2n elements (n any natural number).

The introduction of *derived operators*, like Eventually, supports a modular and intuitive formulation of complex properties. We support in the jABC meanwhile a rich collection of frequently occurring behavioral templates, which ease the declarative formulation of knowledge and goals.

SLTL is a specification language supported by jMosel. In Sect. 4, we will show how we link up to Golog in order to apply it to the mediation problem.

3 Solving the Mediation with M2L(Str) in jMosel

We first formally describe the semantics of service invocation with the associated *use*, *gen* and *kill* sets. At runtime, a service accepts certain inputs and produces certain outputs according to its WSDL description. Semantically, we describe a service (or concrete action) as a transformation on the power set of the set of (concrete) types in the orchestration's context:

$$\mathrm{eff}(\cdot)(\cdot)\colon CA \rightarrow (\mathcal{P}(CT) \rightarrow \mathcal{P}(CT)),$$

Accordingly, a service can only be invoked if all elements in $use(\cdot)$ are available (preconditions), and it produces elements the context according to $gen(\cdot)$, and invalidates the elements of the context specified in $kill(\cdot)$ (effects), thus

$$\mathrm{eff}(a)(T) = \begin{cases} (T \setminus kill(a)) \cup gen(a) & \text{if } use(a) \subseteq T \\ \text{undef} & \text{otherwise} \end{cases} .$$

Incidentally, this is the same abstraction used in Data Flow Analysis to describe the operational semantics of operations, like assignments, which have preconditions and (side) effects. This is the basis for constructing the structure the jMosel synthesis uses as the domain specification, the *configuration universe*, which is rather straightforward, and therefore omitted here due to lack of space.

3.1 jMosel, M2L and SLTL

The search for such a solution is done using a deterministic finite automaton semantically equivalent to the given formula. The input symbols the automaton is fed with are sets of atomic propositions (i.e., types) on one hand and actions (i.e., services) on the other. Since it is not feasible to consider alternating sequences of types and actions, some technical fine-tuning is needed: we now regard *steps*, consisting of an action (or *init*, which is not an action but is used for the head of the path) and a set of atomic propositions. Thus, instead of alternating sequences of sets of types and actions, we now consider strings over the *step alphabet* $\Sigma_{step} = \mathcal{P}(CT) \times (CA \uplus \{init\})$. For example, the path t_0, a_1, t_1, a_2, t_2 now is written as $(t_0, init)(t_1, a_1)(t_2, a_2)$.

There exist several methods to transform an LTL formula into a *Büchi automaton.* However, since we are interested in finite sequences of services rather than infinite words, a Büchi automaton is not quite what we want. One could modify the existing algorithms for generating Büchi automata in the way that NFAs or DFAs are constructed; we, however, want to describe a different approach, namely by using *monadic second-order logic on strings*, M2L(Str) [10], and our toolkit for this logic, jMosel [11,16].

jMosel calculates for a given input formula Φ a deterministic finite automaton A, with $L(\Phi) = L(A)$. A key characteristic of M2L(Str) is that it is *compositional*: atomic formulae are transformed into simple basic automata; for a compound formula, first the automata for its sub formulae are calculated, and then an automata synthesis operation is applied. For details refer to [17].

3.2 Mediator Synthesis

After the domain has been modeled by specifying the relevant modules and the corresponding type and action taxonomies, the mediator synthesis proceeds by the user

- entering a set of initial types T_0. In our example, we start with a *purchase order request*, POR,
- providing the temporal specification formula in SLTL or one of our dialects,
- specifying the desired kind of solution. For the case study, we selected minimality of the solution and the presentation in a format which is directly executable (rather than as a sequence of alternating types and activities), i.e. jABC's SLGs. This results in the same solution as provided via Golog in the next section (cf. Fig. 6.

Tab. 2 presents the corresponding SLTL formula already together with some intuitive explanations. The formula itself, which is just a combination of 4 next-time operators and one eventually operator would easily fit in one line.

Table 2. Explanation of the specification formula used for the jMosel synthesis

Formula element	Explanation
<`ConsumePip3A4POR`>	The synthesized sequence should start with this service.
<`SaveStartTime`>	A local service for time measurement. Since this is local to our requirements, the invocation of this service is required statically by the formula.
F	Find a path in the configuration universe, such that the following two services can be invoked (This is where the actual synthesis action happens).
<`SaveItemsNoAndPOR`>	Again, this is a service local to our requirements, and therefore statically requested.
<`OMServiceV1closeOrder`>	In the end, we want the order to be complete and therefore closed.
tt	Operand, required for the preceding unary `next` operator.

4 The Situation Calculus Solution with Golog

Golog is a high-level programming language based on the situation calculus [12] that was successfully used to solve Web service composition problems [18]. Here we show how to match the problem structure of the mediation scenario with situation calculus, how we generate Basic Action Theories from the domain knowledge representation of Sect. 2.1, how we model an abstract mediation process in Golog, and finally how we synthesize the mediator.

4.1 Intuitive Ontology of the Situation Calculus and Golog

The situation calculus is designed for representing and reasoning about stateful models, dynamically changing worlds, where changes to the world are due to performing named *actions*. S_0 is the initial state of the world, prior to any action. Changes are effected by executing actions in a situation: $s' = do(a, s)$ means that situation s' is obtained by executing the action a in situation s. For example, $do(store(A, B), do(ship(A, B), do(construct(A), S_0)))$ is a situation term denoting the sequence composition $construct(A) \circ ship(A, B) \circ store(A, B)$. Intuitively, situations are action execution paths, and denote indirectly states. Predicates and relations are true in some state and false in others, thus state variables are introduced by means of *relational fluents*. For example, the fluent $location(a, b, s)$ expresses that in the state of the world reached by performing the action sequence s, object a is located at location b.

Domain axiomatizations in the situation calculus are called **Basic Action Theories** and have the form

$$\mathcal{D} = \Sigma \ \cup \ \mathcal{D}_{ss} \ \cup \ \mathcal{D}_{ap} \ \cup \ \mathcal{D}_{una} \ \cup \ \mathcal{D}_{S_0}.$$

where Σ are the foundational axioms for situations, $\mathcal{D}_{ss}$ are the successor state axioms for fluents, $\mathcal{D}_{ap}$ is a set of action precondition axioms for situations, $\mathcal{D}_{una}$ is a set of unique names axioms for actions, $\mathcal{D}_{S_0}$ is a set of first order sentences, uniform in S_0. Basic action theories allow us to reason about action and change, but they offer no way to express a "story" about how certain effects can be achieved, thus no way of expressing procedural knowledge.

The high-level programming language Golog, built on top of Basic Action Theories, allows writing high-level non-deterministic procedures which model such a "story". Golog, in its essential form[4] provides the following language constructs, originally inspired by Algol and its operational semantics:

$Do(\phi?, s, s')$	Test actions
$Do(\delta_1; \delta_2, s, s')$	Sequential composition
$Do(\delta_1 \mid \delta_2, s, s')$	Non-deterministic choice of actions
$Do((\pi x)\delta(x), s, s')$	Non-deterministic choice of action arguments
$Do(\delta^*, s, s')$	Non-deterministic iteration.

[4] Golog has been extended in various ways to include e.g. concurrency, exogenous events, and sensing [12].

that "macro expand" into terms in the situation calculus. As usual, constructs like *if* and *while* can be defined in terms of these basic constructs according to the usual operational semantics. With these constructs, we can write Golog *procedures* specifying the story, or the behavioral knowledge, of a domain and of a solution.

Given a Golog procedure δ, Golog can prove constructively whether it is *executable* with respect to a given Basic Action Theory $\mathcal{D}$: $\mathcal{D} \models \exists s.Do(\delta, S_0, s)$. Since s is the terminating situation of the Golog program δ, the proof returns a sequence of *primitive actions* starting in the initial situation S_0 consistent with the procedure.

The resulting Basic Action Theory $\mathcal{D}_T$ allows us to determine for every situation whether a given activity(service) is executable, and what are the effects of its execution. We now concentrate on the Golog-based process synthesis.

4.2 'Loose' Golog: The :-Operator

In basic Golog, every action needs to be explicitly named in a procedure, and the only operator that allows chaining service executions is the (immediate) successor, corresponding to the SLTL operator Next. In order to specify loose coordinations, that include Eventually and Until, we need operators that allow replacement by an a priori undetermined number of steps. The underlying motivation is that those parts of the concrete processes are unspecified and any sequence that satisfies their boundary conditions is there admissible, building this way a (semantic) equivalence class.

When restricting ourselves to Eventually, which is sufficient for the considered case study, this can be achieved using the :-Operator introduced in [18]. This operator exploits planning/search to achieve the required preconditions of subsequent services by inserting an appropriate sequence of actions. With ':' we can easily specify the mediation task as the Golog procedure *performMediation*:

```
proc $performMediation$
        $consumePip3A4PurchaseOrderRequest(A, B, C, D, E, F, G)$ ;
        $saveStartTime$
:
        $saveItemsNoAndPOR(J, K)$ ;
        $omServiceV1closeOrderSIB(L, M, N)$
endProc.
```

Stating that the mediation process starts with the service *consumePip3A4 PurchaseOrderRequest* and ends with the service *omServiceV1closeOrderSIB*. What happens in between depends on precision of modeling and, in this case, on Golog's search strategy for an action sequence, making the entire sequence executable. The result of the search is a suitable linear, deterministic sequence of actions/activities (a service composition). The services *saveStartTime* and *saveItemsNoAndPOR* are local services which need to be called in order for

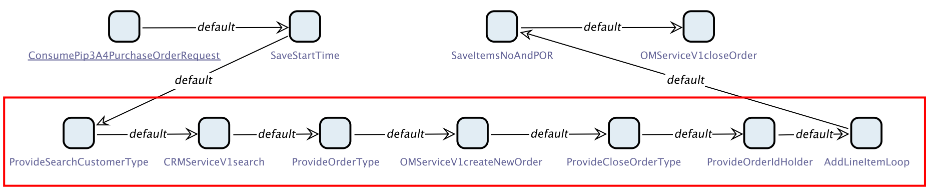


Fig. 6. The resulting Mediator process, visualized as a jABC orchestration. Highlighted in red is the synthesized sequence of actions.

data to be stored on disk for another process (the Mediator part 2, that we do not address here) to be able to retrieve it. The resulting resulting Mediator process is shown in Fig. 6.

5 Conclusion and Perspectives

We have presented two approaches to attack the automatic solution of orchestration problems based on synthesis from declarative specifications, a semantically enriched description of the services, and a collection of services available on a testbed. The first approach uses *jMosel* [11], our tool for solving *Monadic Second-Order Logic on Strings*[19], and the second approach is based on *Golog* [12], a programming language combining imperative features with a Prolog flavor. As a common case study we have considered the *Mediation Scenario* of the *Semantic Web Service Challenge* [1,3], with the goal to synthesize abstract mediator processes, and to describe how the concrete, running (Web) service composition is computed. As a result, together with the solution we had already described in [9,20,21], we have obtained three structurally rather different synthesis solutions (cf. Fig. 2), all integrated as running solutions in the jABC/jETI modeling framework [4,5,6,7].

This could be achieved despite their algorithmic and structural differences:

- our previously exploited method, originally presented in [8], is tableaux-based. It computes the service composition while constructing a proof tree in a fashion reminiscent of the approach presented in [22]
- the jMosel-approach uses a Monadic second-order (M2L) logic-based, compositional automata construction to infer an automaton description from the goal description in SLTL, and
- the Prolog flavored Golog approach synthesizes the mediation process via backward chaining.

There are also some strong similarities:

- All approaches are based on a domain modeling, which essentially consists of the specification of the available services in term of triples that specify the input types, an effect description, and an output type. For the Golog approach, this knowledge is specified within the Situation Calculus in terms

of Basic Action Theories, and in the other two approaches in the so-called Configuration Universe.
- All approaches synthesize an operational description of the mediator from a declarative specification of the available procedural knowledge. In Golog, the declarative description is given in 'loose' Golog, a variant of Golog, resembling an eventuality operator (cf. [18]). The synthesis transforms these loose specifications together with the Basic Action Theories into a concrete runnable Golog program. In the other two approaches the loose descriptions are given in SLTL, a logic specifically designed for temporally loose process specification. The synthesis then results in executable action/module sequences. As can be seen in Fig. 2, as for Golog, the tableaux-based approach directly exploits the domain model, while the M2L-based approach 'projects' the synthesized automaton onto the Configuration universe via simple product construction.
- From a semantical perspective, all approaches use a variant of (Kripke) Transitions Systems (KTS) [23] as their operational model, i.e. kinds of automata, where the edges are labelled with actions/activities, and where the nodes (implicitly) resemble type information/fluents. The fact that Golog is intuitively linked here to a tree structure rather than to a graph structure is technically of minor importance. However, here the fact transpires that Golog is intend to construct plans (essentially paths in a tree) essentially instance-driven, rather than to represent potentially all possible plans, as it is possible with the other approaches considered here.
- All approaches have a computational bottleneck: for the tableaux method it is the explosion of the proof tree, M2L-synthesis is known to be non-elementary (the automata construction may explode due to the required intermediate determination), and also backward chaining is classically known to be a hard problem. It is our goal to help understanding which bottleneck strikes when, and where to prefer which (combination of) which technologies.

This similarity/difference spectrum is ideal for an investigation of application profiles. We are therefore currently investigating *when which bottleneck strikes, and why.* Please note that there may be significant differences here between superficially similar approaches, with drastic effects on the performance depending on the considered situation. For comparison, consider the situation in Model Checking, where techniques like (BDD-based) symbolic model checking, bounded model checking, assume-guarantee techniques, partial order reduction, etc., are known to cover very different aspects of the so-called state explosion problem. Even in this well-studied field these effects are still not fully understood.

This is the motivation for us to work on a common platform for experimentation, where the different techniques can be evaluated, compared, modified and combined. Technological basis for this is the jABC/jETI modeling and experimentation framework [4,5,6,7], which has been initiated more than 10 years ago [14], and which has shown its power in other contexts, see e.g. our model learning environment [24,25]. At the moment four approaches have been integrated, and we plan to integrate more within the next year, in particular one exploiting traditional automata-theoretic methods for translation LTL to Büchi automata (cf. eg. [26]), one following the idea of Hierarchical Task Networks (cf. eg. [27]), which exploit given knowledge about task decomposition, and also the input/output function-focussed approach based on the Structural Synthesis Program described in [28]. This way we do not only want to be able to fairly compare the different scenarios in different contexts for their application profiles, but also to enlarge the landscape of Fig. 2 to a library of powerful synthesis and planning components which can be combined within jABC to new complex domain-specific planning or synthesis solutions. In order to make this possible at a larger scale, we plan to make the experimentation platform publicly available, allowing people not only to experiment with the integrated tools, but also to provide their own tools for others to experiment with. We hope that this will contribute to a better experimentation-based understanding of the various methods and techniques, and a culture of systematic application-specific construction of synthesis solutions.

References

1. Semantic Web Service Challenge (2009), `http://www.sws-challenge.org`
2. RosettaNet standard (2009), `http://www.rosettanet.org/`
3. Petrie, C., Margaria, T., Lausen, H., Zaremba, M. (eds.): Service-oriented Mediation with jABC/jETI. Springer, Heidelberg (2008)
4. Jörges, S., Kubczak, C., Nagel, R., Margaria, T., Steffen, B.: Model-driven development with the jABC. In: HVC - IBM Haifa Verification Conference, Haifa, Israel, IBM, October 23-26, 2006. LNCS. Springer, Heidelberg (2006)
5. Steffen, B., Margaria, T., Braun, V.: The electronic tool integration platform: Concepts and design. Int. Journal on Software Tools for Technology Transfer (STTT) 1(2), 9–30 (1997)
6. Margaria, T.: Web services-based tool-integration in the ETI platform. SoSyM, Int. Journal on Software and System Modelling 4(2), 141–156 (2005)
7. Steffen, B., Margaria, T., Nagel, R.: Remote Integration and Coordination of Verification Tools in jETI. In: Proc. of ECBS 2005, 12th IEEE Int. Conf. on the Engineering of Computer Based Systems, Greenbelt (USA), April 2005, pp. 431–436. IEEE Computer Society Press, Los Alamitos (2005)
8. Freitag, B., Steffen, B., Margaria, T., Zukowski, U.: An approach to intelligent software library management. In: Proc. 4th Int. Conf. on Database Systems for Advanced Applications (DASFAA 1995), National University of Singapore, Singapore (1995)
9. Margaria, T., Bakera, M., Kubczak, C., Naujokat, S., Steffen, B.: Automatic Generation of the SWS-Challenge Mediator with jABC/ABC. Springer, Heidelberg (2008)
10. Church, A.: Logic, arithmetic and automata. In: Proc. Int. Congr. Math.,Uppsala, Almqvist and Wiksells, vol. 1963, pp. 23–35 (1963)
11. Topnik, C., Wilhelm, E., Margaria, T., Steffen, B.: jMosel: A Stand-Alone Tool and jABC Plugin for M2L(str). In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 293–298. Springer, Heidelberg (2006)
12. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press, Cambridge (2001)

13. Steffen, B.: Generating data flow analysis algorithms from modal specifications. Sci. Comput. Program. 21(2), 115–139 (1993)
14. Steffen, B., Margaria, T., Braun, V.: The electronic tool integration platform: Concepts and design. Int. Journal on Software Tools for Technology Transfer (STTT) 1(2), 9–30 (1997)
15. Margaria, T., Steffen, B.: LTL guided planning: Revisiting automatic tool composition in ETI. In: SEW 2007: Proceedings of the 31st IEEE Software Engineering Workshop, Washington, DC, USA, pp. 214–226. IEEE Computer Society Press, Los Alamitos (2007)
16. Wilhelm, C.T.E., Steffen, T.M.B.: jMosel: A stand-alone tool and jABC plugin for M2L(Str). In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 293–298. Springer, Heidelberg (2006)
17. Margaria, T.: Fully automatic verification and error detection for parameterized iterative sequential circuits. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 258–277. Springer, Heidelberg (1996)
18. McIlraith, S., Son, T.: Adapting golog for composition of semantic web services. In: Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002), Toulouse, France, April 22-25, 2002, pp. 482–493 (2002)
19. Kelb, P., Margaria, T., Mendler, M., Gsottberger, C.: MOSEL: A flexible toolset for monadic second-order logic. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 183–202. Springer, Heidelberg (1997)
20. Kubczak, C., Margaria, T., Kaiser, M., Lemcke, J., Knuth, B.: Abductive synthesis of the mediator scenario with jABC and GEM. Technical Report LG-2009-01, Stanford University (2009), `http://logic.stanford.edu/reports/LG-2009-01.pdf`
21. Lemcke, J., Kaiser, M., Kubczak, C., Margaria, T., Knuth, B.: Advances in solving the mediator scenario with jABC and jABC/GEM. Technical Report LG-2009-01, Stanford University (2009), `http://logic.stanford.edu/reports/LG-2009-01.pdf`
22. Baier, J., McIlraith, S.: Planning with temporally extended goals using heuristic search. In: Proc. ICAPS 2006, Cumbria, UK. AAAI, Menlo Park (2006)
23. Müller-Olm, M., Schmidt, D.A., Steffen, B.: Model-checking: A tutorial introduction. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 330–354. Springer, Heidelberg (1999)
24. Raffelt, H., Steffen, B., Berg, T.: Learnlib: a library for automata learning and experimentation. In: Proc. of ACM SIGSOFT FMICS 2005, pp. 62–71. ACM Press, New York (2005)
25. Margaria, T., Raelt, H., Steen, B., Leucker, M.: The learnlib in FMICS-jETI. In: Proc. of ICECCS 2007, 12th IEEE Int. Conf. on Engineering of Complex Computer Systems, July 2007. IEEE Computer Soc. Press, Los Alamitos (2007)
26. Gastin, P., Oddoux, D.: Fast ltl to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 53. Springer, Heidelberg (2001)
27. Sirin, E., Parsia, B., Wu, D., Hendler, J.A., Nau, D.S.: HTN planning for web service composition using shop2. In: ISWC 2003, vol. 1(4), pp. 377–396 (2003)
28. Matskin, M., Rao, J.: Value-added web services composition using automatic program synthesis. In: Bussler, C.J., McIlraith, S.A., Orlowska, M.E., Pernici, B., Yang, J. (eds.) CAiSE 2002 and WES 2002. LNCS, vol. 2512, pp. 213–224. Springer, Heidelberg (2002)