Semantic Web Services Challenge 2008

Synthesizing the Mediator with jABC/ABC

Tiziana Margaria
Chair of Service and Software Engineering, Universität Potsdam (Germany)
margaria@cs.uni-potsdam.de

Marco Bakera, Harald Raffelt, Bernhard Steffen
Chair of Programming Systems, TU Dortmund (Germany)
{marco.bakera, harald.raffelt, steffen}@cs.uni-dortmund.de

Abstract. In this paper we show how to apply a tableau-based software composition technique to automatically generate the mediator's service logic. This uses an LTL planning (or configuration) algorithm originally embedded in the ABC and in the ETI platforms. The algorithm works on the basis of the existing jABC library of available services (SIB library) and of an enhanced description of their semantics given in terms of a taxonomic classification of their behaviour (modules) and abstract interfaces/messages (types).

1 The SWS Challenge Mediator

The ongoing Sematic Web Service Challenge [19] proposes a number of increasingly complex scenarios for workflow-based service mediation and service discovery. We use here the technology presented in [10] to synthesise a process that realizes the communication layer for the Challenge's initial mediation scenario.

In this scenario, a customer (technically, a client) initiates a Purchase Order Request specified by a special message format (RosettaNet PIP3A4) and waits for a corresponding Purchase Order Confirmation according to the same RosettaNet standard. The seller however does not support this standard. Its backend system or server awaits an order in a proprietary message format and provides appropriate Web Services to serve the request in the proprietary format. As client and server here speak different languages, there is a need for a mediation layer that adapts both the data formats and also the granularity.

Of course we can easily define the concrete process within our jABC modelling framework, as we have shown in the past [11, 6, 7].

To provide a more flexible solution framework, especially to accommodate later *declarative specification changes* on the backend side or on the data flow, we synthesize the whole mediator using the synthesis technology introduced in [10]. We proceed here exactly along the lines already presented in that paper.

In the following, we show in Sect. 2 how to use the SLTL synthesis methodology to generate the mediator workflow based on a knowledge base that expresses the semantics

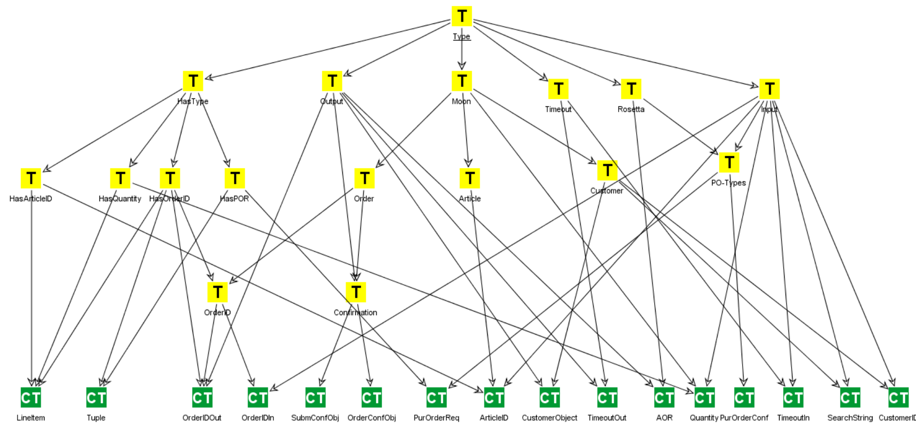

Fig. 1. The SWS Challenge Mediator Type Taxonomy

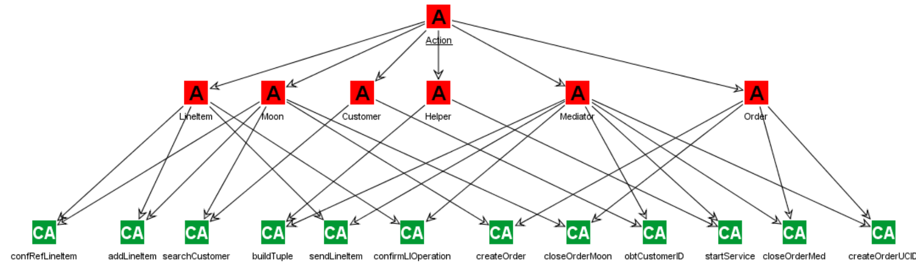

Fig. 2. The SWS Challenge Mediator Module Taxonomy

of the concrete types from the SWS mediator scenario, then in Sect. 3 we add a more business-level-like abstraction to the knowledge base,

and in Sect. 4 we show how this leads to a looser solution, and how this solution can be stepwisely refined towards the first solution by adding business-level knowledge to the problem definition, in a declarative way. Sect. 5 describes how to work with the synthesis tool. Finally, Sect. 6 discusses related work and Sect. 7 draws some conclusions and sketches ongoing work.

2 The Concrete Mediator Workflow

2.1 Abstract Semantics: Taxonomies for Modules and Types

Table 1 shows the modules identified within the system. They represent at the semantic level the collection of basic services available for the mediator. In order to produce a running solution as demonstrated in Stanford in November they are then bound (grounded) to the concrete SIBs that in the jABC constitute the running services. How

module name	input type	output type	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

Table 1. The SWS mediation Modules

this happens is sketched in [17].

This information about the single modules is complemented by simple ontologies that express in terms of *is-a* and *has-a* relations properties over the types and the modules of the scenario. We call these relations Taxonomies. The taxonomies regarding the mediation scenario are shown in Fig. 1 (Type Taxonomy) and Fig. 2 (Module Taxonomy).

This information is expressed in a Prolog-like fashion in a concrete knowledge base which feeds then the synthesys algorithm.

2.2 The Concrete Knowledge Base

The synthesis tool takes as input a textfile with the definitions of the taxonomies (module and type taxonomy), the module descriptions, and some documentation. The first line of the file declares a name for the knowledge base:

```
$program(sws_challenge).
```

The file contains statements (one per line) of facts in the following three forms:

- `tax(type, output, customerObject).`
- `tax(module, mediator, sendLineItem).`
- `module(searchCustomer, searchString, customerObject).`

The two first statements show how to specify the type and module taxonomy:

- The first line declares `customerObject` as a subtype of the `output` type.
- The second line declares module `sendLineItem` to be a `mediator` module.

The third statement form is used to specify the relation between input and output types for particular modules. It describes the module definition as already presented in Table 1: the `searchCustomer` module takes a `searchString` as input type and produces a `customerObject` output type.

This way it is possible to concisely represent the taxonomies of Fig. 1 and 2 as well as the module description of Table 1 in one single file.

2.3 Semantic Linear-time Temporal Logic

The loose specification language supported by the synthesis is the *Semantic Linear-time Temporal Logic* (SLTL)[14], a temporal (modal) logic comprising the taxonomic specifications of types and activities. This lifts the classical treatment of types and activities in terms of actions and propositions to a semantical level in a way typical today in the context of the semantic Web.

Definition 1 (SLTL).

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

$$\Phi ::= \ \mathsf{type}(t_c) \mid \neg\Phi \mid (\Phi \wedge \Phi) \mid <a_c> \Phi \mid \mathbf{G}(\Phi) \mid (\Phi \mathbf{U} \Phi)$$

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

SLTL formulas are interpreted over the set of all *legal coordination sequences*, i.e. alternating type correct sequences of types and activities[1], which start and end with types. The semantics of SLTL formulas is now 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 $<>$:
  $<a_c> \Phi$ is satisfied by coordination sequences whose second element (the first activity) satisfies a_c and whose *continuation*[3] satisfies Φ. In particular, $<tt> \Phi$ is satisfied by every coordination sequence whose continuation satisfies Φ.
- Generally operator **G**:
  $\mathbf{G}(\Phi)$ requires that Φ is satisfied for every suffix[4] satisfies Φ.

---

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

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

3 This continuation is simply the coordination sequence starting from the third element.

4 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).

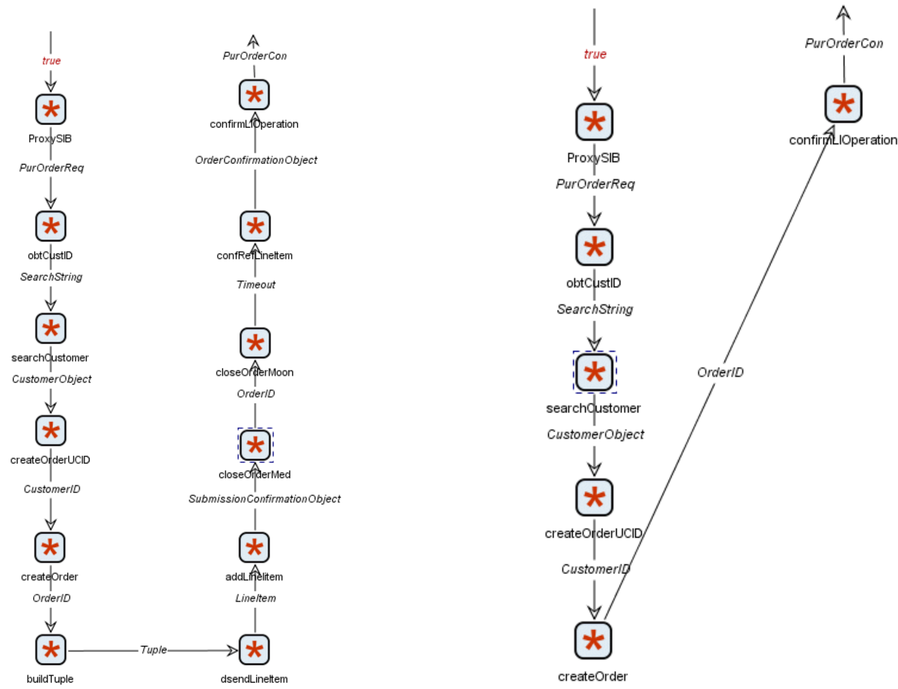

Fig. 3. (a) The synthesised SWS mediator (standard) and **(b)** Using loose types: the new solution

- Until operator **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 continuation satisfies the property Ψ. Note that $\Phi \mathbf{U} \Psi$ guarantees that the property Ψ holds eventually (strong until).

The definitions of continuation and suffix may seem complicated at first. However, thinking in terms of path representations clarifies the situation: a subpath always starts with a node (type) again. Users should not worry about these details: they may simply think in terms of pure activity compositions and not care about the types, unless they explicitly want to specify type constraints.

The online introduction of *derived operators* supports a modular and intuitive formulation of complex properties.

2.4 Declarative LTL Specification for the Concrete Mediator

For the mediator, we look for a workflow (a service coordination) that satisfies the following requirement:

The mediator service should produce a Purchase Order Confirmation.

The corresponding formal specification formulated in SLTL is simple: we need to start the service (module startService) and reach the result `PurOrderCon` (a type). We

may simply write: `(startService < PurOrderCon)` where the symbol < denotes a derived operator meaning *before* or *preceeds* and is defined as

$$\mathtt{f1} < \mathtt{f2} =_{df} \mathbf{F}(\mathtt{f1} \wedge \mathbf{F}(\mathtt{f2}))$$

The jABC process model shown in Fig. 3(a) resembles very closely the expected required solution.

If we adopt the very fine granular model of the types shown in Table 1, a natural choice given the SWS Challenge problem description, this is in fact the only solution.

In this setting, we use abstract type names in the taxonomy to model de facto almost the concrete operational semantics: we distinguish for instance an `OrderID` from an `OrderConfObject`, modelling the described application domain at the concrete level of datatypes and objects - a direct rendering of what happens at the XML level, or for programs in the memory and in the heap. This is however already a technical view, and it corresponds to lifting the concrete, programming-level granularity of data to the semantic level: the resulting ontology is as concrete as the underlying program.

This is however not the intention of Service Orientation, nor of the semantic web: the idea there is to decouple the business-level view (captured at the semantic level) from the technical view of a specific implementation, in order to allow a coarser description of business-level workflows and processes that then must be concretized and grounded to a running implementation. In the following we show how this can be done, also including automatic synthesis.

3 Abstract Semantics: Using Abstraction and Constraints

For a specifier and definer of the business domain it is much more realistic to say that the modules concerned with orders work on an `Order` type, which is a business-level abstraction for order-related objects and records, and to leave the distinctions to a problem-specific refinement of the desired solutions via constraints added at need.

For the abstract semantics we work on the taxonomies. The taxonomy design and module specification decides here the balance between concreteness and flexibility (looseness). In this specific case, we change the definition of the modules that deal with orders as shown in Tab. 2: they now operate on the abstract `Order` type. We can be as concrete, or as abstract and generic as we wish, and choose the suitable description level driven by the semantics or application domain modelling. This abstraction determines how much flexibility we build in into our solutions. At the one extreme we can have very specific types, as fine granular as a description in terms of structural operational semantics [12]. In this case, solutions are type-determined, and basically render the concrete labelled transition system underlying the manually programmed solution as in Fig. 3(a). At the other extreme one could also model the process structure solely by means of temporal constraints. However, most flexible is a hybrid approach which combines loose taxonomies and module descriptions with temporal constraints in order to arrive at an adequate specification formalism.

No matter the choice, the algorithm covers the whole spectrum, leaving it free to the application domain designer to determine where to be precise and where to be loose, leaving space for exploring alternatives and tradeoffs.

module name	input type	output type	description
Mediator			Maps RosettaNet messages to the backend
buildTuple	*Order*	*Tuple*	Builds a tuple from the orderID and the POR
closeOrderMed	*SubmConfObj*	*Order*	Closes an order on the mediator side
confirmLIOperation	*Order*	*PurOrderCon*	Receives a conf. or ref. of a LineItem and sends a conf.
Moon			The backend system
createOrder	*CustomerID*	*Order*	Creates an order
closeOrderMoon	*Order*	*TimeoutOut*	Closes an order on the backend side
confRefLineItem	*Timeout*	*Order*	Sends a conf. or ref. of a prev. subm. LineItem

Table 2. The SWS Mediation Modules with abstract Order

4 A Loose Solution, and its Declarative Refinement

4.1 The base case

If we now solve the planning problem with the modified module description and the original goal, we obtain a much shorter solution, shown in Fig. 3(b). This is due to the fact that these module specifications now refer to the abstract type *Order*. As a consequence, closeOrderMoon is a suitable direct successor of createOrder. This solution corresponds to a degenerate workflow where an empty order is sent.

4.2 Refinement1: Nonempty Orders

Since in the normal case orders contain items, the business expert needs to be more precise in the specification of the solution, adding knowledge by means of SLTL constraints. If one just knows that the items are referred to via the LineItem type, one may simply refine the goal as follows:

```
(startService < LineItem < PurOrderCon)
```

This way, we have added as additional intermediate goal the use of a LineItem type. Accordingly, at least one of the modules {addLineItem, sendlineItem} must appear in the required minimal workflow. We see the result in Fig. 4(a): this solution coincides with the previous one till the createOrder module, then the type mediator buildTuple is added, after which sendLineItem satisfies the intermediate goal. The remaining constraint at that point is simply the reaching of the final type PurOrderCon, which is done by generating the sequence CloseOrderMediator followed by CloseOrder.

This solution however corresponds only to the first Web service realizing the mediator. There is in fact a subsequent second service that realizes the confirmation part of the mediator.

4.3 Refinement2: Confirmed Nonempty Orders

To obtain this part as well, we have to additionally specify that we need to see a confirmation, e.g. as confRefLineItem module:

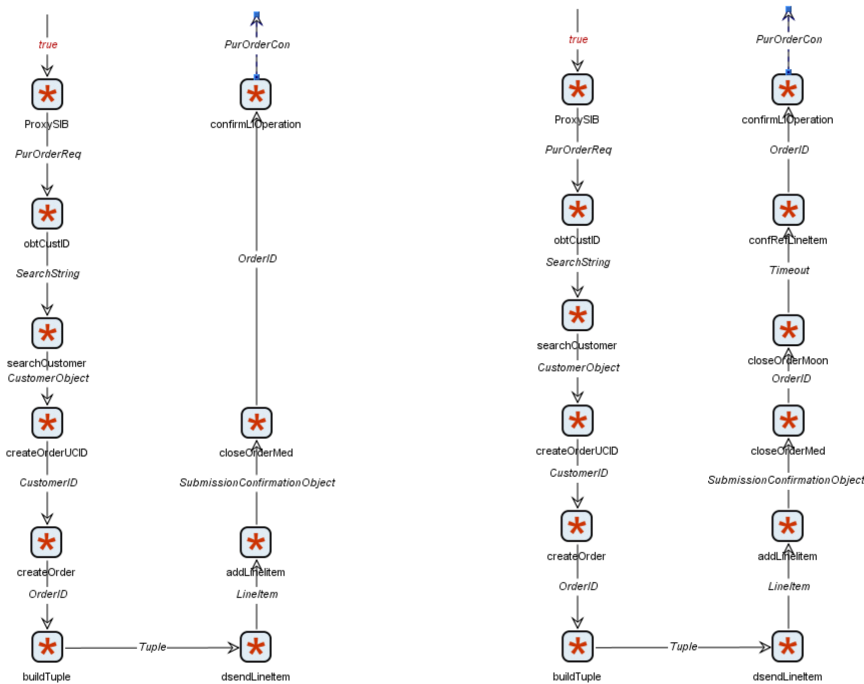

Fig. 4. (a) Adding a LineItem: the new solution and **(b)** Adding a Confirmation: the complete loose solution

```
(startService < LineItem <
      confRefLineItem <PurOrderCon)
```

This generates the solution of Fig. 4(b), which includes also the rest of the sequence shown in Fig. 3(a).

5 How to work with the Synthesis Tool

The synthesis tool takes as input the text file containing the knowledge base: the module and type taxonomy, the module descriptions, and some documentation for the integrated hypertext system. It is steered from the ABC GUI. There, users can input the SLTL formulas that describe the goal and can ask for different kinds of solutions. The tool produces a graphical visualization of the satisfying plans (module compositions), which can be executed, if the corresponding module implementations are already available, or they can be exported for later use.

The knowledge basis implicitly describes the set of all legal executions. We call it *configuration universe*, and it contains all the compatible module compositions with respect to the given taxonomies and to the given collection of modules. Fig. 5 shows the configuration universe that emerges when simply taking the atomic, concrete input/output types.

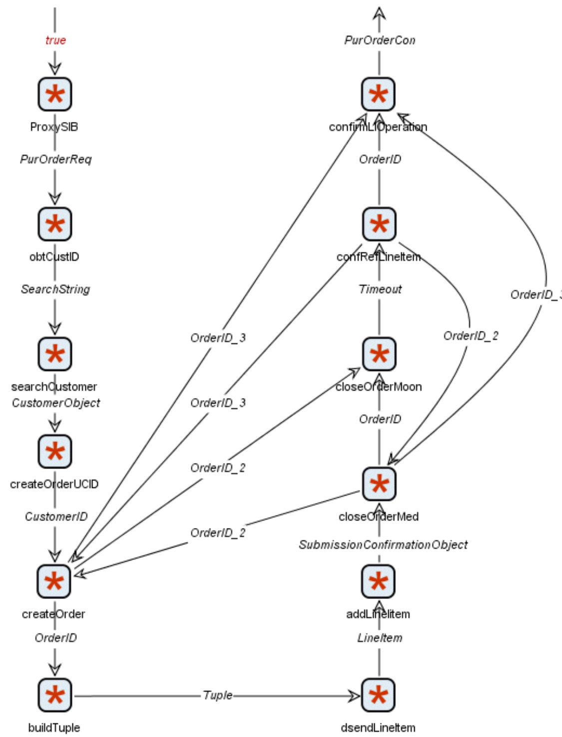

Fig. 5. The Configuration Universe

5.1 Specifying Solution Types

Users never see the configuration universe. They have a number of simple options to state which kind of solutions they would like to have displayed.

- **minimal** solutions denotes plans that achieve the goal without repetition of configurations. In particular, this excludes cycles.
- **shortest** solutions returns the set of all minimal plans that are also shortest, measured in number of occurring steps.
- **one shortest** solution returns the first shortest plan satisfying the specification.
- **all** solutions returns all the satisfying solutions, which includes also cyclic ones.

Minimal plans generated for our working example are shown in Fig. 6. Since these plan descriptions are directed acyclic graphs, it is rather simple to select and execute one plan.

The typical user interaction foresees a successive refinement of the declarative specification by starting with an initial, intuitive specification, and asking typically for shortest or minimal solutions, and using the graphical output for inspection and refinement.

This is exactly what we did in Sect. 3, where we used abstract types to enlarge the solution space and then tightened successively the LTL specification by adding salient characteristics that yield a good declarative characterization of the desired solutions.

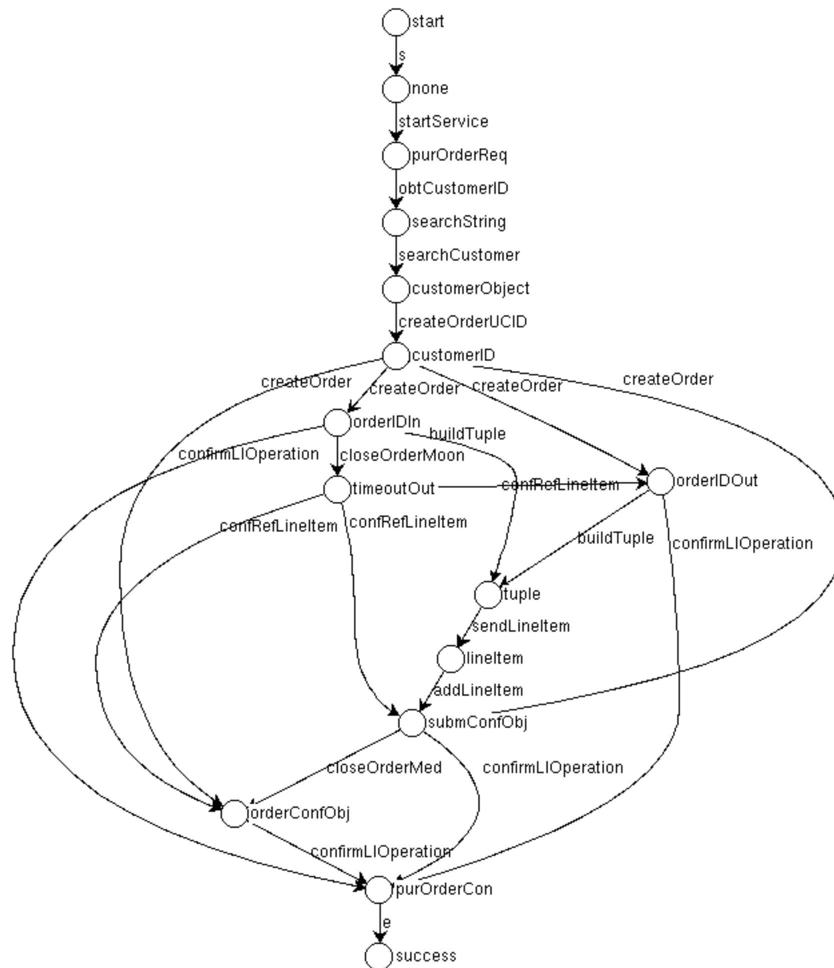

Fig. 6. The Minimal Solutions

6 Related Approaches

Our approach was introduced 1993 ins [15, 4] and applied in [16, 9] and [18] to synthesize Statecharts, CAD design processes, and heterogeneous verification algorithm for concurrent systems, respectively. The idea of LTL guided process composition has later been taken up by others: Bacchus and Kabanza [1]extensively discuss their technique that implements a first order extension of LTL, Mandell and McIlraith use LTL in the context of BPEL compositions [8], and Falcarin et al. [20] use LTL as a starting point for their compositions, transforming single LTL formulas to finite state automata, then composing them to a global specification, and finally finding the correct shortest solutions as the acyclic accepting paths in that automaton.

Concerning the relation with planning, the state variables in an LTL formula are directly fluents: their value changes from state to state along the process, and the formulas describe mutual dependencies naturally and compactly. In this sense, there is a close kinship between the temporal logic mentality and event calculus [13] or logics for timing diagrams [3]: all three describe what is true at what time, associating the evolution of time with a succession of states, and offering a well chosen set of operators to express dependencies between temporal variables along possible paths within models. The fundamental advantages of LTL guided synthesis over planning are the following:

- the guidance it allows is process driven and not state driven. Therefore the control it offers can in general depend on the entire history of predecessors, and not only on the current state. This is extremely efficient in focussing the search, resulting in small memory usage and quick execution.
- it is decoupled from the (internal) state of a solver/planner: the search control information relates exclusively to properties of the domain knowledge, not on any information on the internal state of an algorithm, which is often the case for planning techniques in order to capture and encode the relevant history aspects (what is enabled, what is true, etc.) that govern the correct chaining of transitions, i.e. the temporal/causal/precedence aspects. In contrast, a user of our technique does not need to know anything about the algorithm underlying the solver/planner.

7 Conclusions

We have applied the automatic tool composition feature of the ABC/ETI platform as a synthesis tool for the mediator. Our LTL-based synthesis approach is not restricted to compute one solution, but it may compute all (shortest/minimal) solutions, with the intent to provide maximum insight into the potential design space.

In future we plan to investigate various forms of synthesis approaches in order to compare their application profiles. In particular, we are interested in comparing game-based methods which work via synthesis of winning strategies with the described tableau-based methods, that construct a linear model as a result of proof construction. We also plan to enhance the user-friendliness in terms of graphical support for the declarative specifications, for example by means of the Formula Builder [5] and by the use of patterns [2].

References

1. F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123 – 191, 2000.
2. M. Dwyer and J.Corbett G. Avrunin. *Specification Patterns Website*. `http://patterns.projects.cis.ksu.edu/`.
3. Kathi Fisler. Toward diagrammability and efficiency in event-sequence languages. *STTT, Int. J. on Software Tools for Technology Transfer*, 8(4-5):431–447, 2006.
4. B. Freitag, B. Steffen, T. Margaria, and U. Zukowski. An approach to intelligent software library management. In *Proc. 4th Int. Conf. on Database Systems for Advanced Applications (DASFAA '95), National University of Singapore, Singapore*, 1995.

5. S. Jörges, T. Margaria, and B. Steffen. Formulabuilder: A tool for graph-based modelling and generation of formulae. In *Proc. ICSE'06*, May 2006.
6. C. Kubczak, T. Margaria, B. Steffen, and S. Naujokat. Service-oriented mediation with jeti/jabc: Verification and export. In *Worksh. on Service Composition & SWS Challenge, part of WI-IAT'07, the IEEE/ WIC/ ACM Int. Conf. on Web Intelligence, November 2007, Stanford (CA)*, volume ISBN-10: 0-7695-3028-1. IEEE CS, 2007.
7. C. Kubczak, T. Margaria, C. Winkler, and B. Steffen. An approach to discovery with miaamics and jabc. In *Worksh. on Service Composition & SWS Challenge, part of WI-IAT'07, the IEEE/ WIC/ ACM Int. Conf. on Web Intelligence, November 2007, Stanford (CA)*, volume ISBN-10: 0-7695-3028-1. IEEE CS, 2007.
8. Daniel J. Mandell and Sheila A. McIlraith. Adapting bpel4ws for the semantic web: The bottom-up approach to web service interoperation. In *Proc. ISWC2003, Sundial Resort, Sanibel Island, FL (USA), LNCS N.2870, 2003, pp. 227 - 241, Springer Verlag*, 2003.
9. T. Margaria and B. Steffen. Backtracking-free design planning by automatic synthesis in metaframe. In *Proc. FASE'98, Lisbon(P), LNCS, Springer Verlag*, 1998.
10. T. Margaria and B. Steffen. LTL guided planning: Revisiting automatic tool composition in ETI. In *SEW: 31st Annual Software Engineering Workshop*. IEEE Computer Society Press, March 2007.
11. T. Margaria, C. Winkler, C. Kubczak, B.Steffen, M. Brambilla, D. Cerizza S. Ceri, E. Della Valle, F. Facca, and C. Tziviskou. The sws mediator with webml/webratio and jabc/jeti: A comparison. In *Proc. ICEIS'07, 9th Int. Conf. on Enterprise Information Systems*, Funchal (P), June 2007.
12. G.D. Plotkin. *a structural approach to operational semantics.* Journal of Logic and Algebraic Programming.
13. *M. Shanahan. The event calculus explained. In* LNAI (1600):409-430*. Springer Verlag, 1999.*
14. *B. Steffen, T. Margaria, and A. Claßen.* heterogeneous analysis and verification for distributed systems. *SOFTWARE: Concepts and Tools*, 17(1):13–25, 1996.
15. B. Steffen, T. Margaria, and B. Freitag. Module configuration by minimal model construction. In *Tech. rep. MIP 9313, Universität Passau, Passau (D)*, 1993.
16. B. Steffen, T. Margaria, and M. von der Beeck. Automatic synthesis of linear process models from temporal constraints: An incremental approach. In *Proc. AAS'97, ACM/SIGPLAN Int. Workshop on Automated Analysis of Software, Paris (F),(affiliated to POPL'97), pp. 127-141.*, 1997.
17. B. Steffen and P. Narayan. *full lifecycle support for end-to-end processes.* IEEE Computer*, 40(11):64–73, Nov., 2007.*
18. *Bernhard Steffen, Tiziana Margaria, and Ralf Nagel. Remote Integration and Coordination of Verification Tools in jETI. In* Proc. ECBS 2005, 12th IEEE Int. Conf. on the Engineering of Computer Based Systems*, pages 431–436, Greenbelt (USA), April 2005. IEEE Computer Soc. Press.*
19. SWS Challenge Workshops: Website*. http://sws-challenge.org/wiki/index.php/Workshops.*
20. *J. Yu, J. Han, Y. Jin, and P. Falcarin. Synthesis of service compositions process models from temporal business rules.*