HarmonICS - a Tool for Composing Medical Services [⋆]

Dariusz Doliwa[1], Wojciech Horzelski[1], Mariusz Jarocki[1], Artur Niewiadomski[2], Wojciech Penczek[2,3], Agata Półrola[1], and Jarosław Skaruz[2]

[1] University of Łódź, FMCS, Banacha 22, 90-238 Łódź, Poland
{doliwa,horzel,jarocki,polrola}@math.uni.lodz.pl
[2] Siedlce University of Natural Sciences and Humanities, ICS,
3-go Maja 54, 08-110 Siedlce, Poland
{artur,jskaruz}@ii.uph.edu.pl
[3] Institute of Computer Science, PAS, Ordona 21, 01-237 Warsaw, Poland
penczek@ipipan.waw.pl

Abstract. The paper presents the tool HarmonICS designed for automated composition of medical services and implementing our approach to description and composition of web services. HarmonICS enables arranging sequences of services to satisfy a user's request specified by a query. The query language is rich enough to express requirements on the timing and the ordering of services used.

1 Introduction and Related Work

We present a new tool for automated composition of web services (WS) related to the medical domain. The tool implements our original approach [7] to WS composition, based on introducing a uniform semantic description of services, an object model for the problem, and applying a multi-phase composition supported by model checking methods. The planning process aims at satisfying a user's goal, specified in a declarative language, which enables not only to express features of the objects, but also requirements on the timing and ordering of services occurring in the plan.

The WS composition problem is a very important subject of research for which many various solutions exist. The simplest ones are based on explicit state space search algorithms [16], while more advanced ones employ a graph-based planning [5], logic programming [14], an AI planning [13, 11], model checking methods [10, 12], and genetic algorithms [3]. Vitvar et al. [17] proposed a solution based on WSMO/WSML [15] formalisms. While the fundamental ideas of their concepts seem similar to ours, it is important to mention that our approach is simpler and thus much easier to implement.

Our considerations follow that of Ambroszkiewicz [1], which provides a specification of an automatic composition system based on a multi-phase composition and uniform semantic descriptions of services. However, several extensions like enriched descriptions of services or a hierarchic organisation of services and objects they operate on, have been additionally designed. Doing all that, we keep the semantic base as simple as possible, which aims at enabling a translation of the WS composition problem to a problem solvable by means of eficient methods and tools from other domains.

[⋆] Partly supported by National Science Centre under the grant No. 2011/01/B/ST6/01477.

The first "general" implementation of our approach (system PlanICS) was described by Doliwa et al. [8]. The tool HarmonICS to be presented here is, on one hand, an extension of PlanICS due to incorporating new theoretical solutions, while on the other hand it is its specialization to a particular domain. In addition to the SAT-based planning method inherited from PlanICS, HarmonICS offers also a new specialized SMT-based solution. The SMT-based concrete planner has been developed in response to insufficient performance of the previous solution in some particular cases. The bottleneck was a translation of TADDPA[1] to SAT in the presence of a large number of conditions imposed on the variables, especially these "expensive" ones, e.g., using modulo operator.

In addition to introducing the SMT-based planner, our contribution consists in developing several extensions of the underlying formalisms, which are discussed in the next section together with the theoretical background of our approach. The rest of the paper is organizes as follows. Sec. 3 introduces the main features of our solution, and gives an overview of the system implementation. Finally, a summary and a comparison between HarmonICS and PlanICS are provided in Sec. 4.

2 Theory behind HarmonICS

Our approach to automated composition of WSs is based on introducing a unified semantics for functionalities offered by services. A service is understood as a function which transforms a set of data into another set of data. The sets of data, i.e., inputs and outputs of services, are described in terms taken from a "dictionary" of types, introduced by an appropriate *ontology*. Each ontology follows the standard object model with classes, objects as their instantiations, and attributes as their components. More precisely, both the services and the items they operate on are organised into a multiple inheritance hierarchy of types, the top of which is composed of the following classes: *Thing* of no attributes and its descendants: *Object*, *Service*, and *Trace*.

Below we explain the meaning of the branches rooted at the three descendant classes of *Thing* mentioned above. An example fragment of the ontology tree is presented in Fig. 1, where the solid arrows stand for the inheritance relation. We embed our explanation in the context of medical services considered in the paper. Therefore, we use the names from Fig. 1 as examples, but, in fact, all the nodes below *Object*, *Service*, and *Trace* are **domain-dependent**, and even for a "fixed" domain they can vary depending on the modelling assumed.

The branch of classes rooted at *Object* introduces "types of beings", *Patient, Diagnosis, Therapy*, that are necessary to specify what the services operate on, together with the "features" of these beings expressed by their attributes. For example the class *Patient* has the attributes *First_name*, *Last_name*, *Address*, *Date_of_Birth*, *Diagnosis* etc. having a clear intuitive meaning.

The branch of classes rooted at *Service* introduces types of services - *Visit, Treatment, Registration*. The attributes of the class *Service*, inherited by all its descendants, are as follows : *in*, *out*, *inout*, *preCondition*, and *postCondition*. The first three of them are aimed at listing objects (classified by names and types, similarly to subprogam parameters) which, respectively, are required to execute the service (*in*), are produced by

[1] Timed Automata with Discrete Data and Parametric Assignments

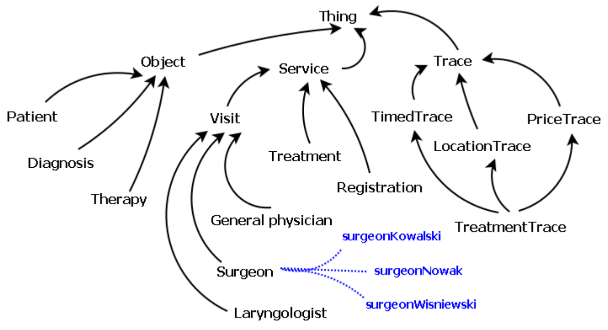


Fig. 1. A fragment of the ontology used by HarmonICS

the service (*out*), and are taken as an input and returned modified (*inout*). The aim of *preCondition* and *postCondition* is to specify respectively the conditions which should be satisfied by the "input" objects to have the service started and the conditions the "output" object satisfy after the service has been executed. For example we can express that the services of the type *Visit* modify an instance of *Patient* by placing *p:Patient* in the *inout* list, and require a visit to result in a diagnosis by placing *isSet(p.Diagnosis)* in the *postCondition*. The values of the attributes common for all the services of a given type are specified in a special instance of the corresponding class, called an *abstract service*. The *concrete services* of a given type (instances of the class representing this type) can introduce their own extensions to the attributes above. For example, the concrete service *GeriatricianSmith* of type *GeriatricianVisit* can require his patients to be older than 85 by extending the common *preCondition* by *p.Date_of_Birth* < *"1927-12-31"*). A more detailed description of the above elements of ontologies can be found in [7].

A new concept introduced to HarmonICS is shown as the third branch (from the left) of the inheritance tree in Fig. 1, i.e., the class *Trace* and its descendants. The instances of the above classes, called *Traces*, are "virtual products" (not corresponding to real-world beings). The *out* list of each service contains exactly one element corresponding to a trace, e.g. *t:Trace*. The main motivation behind *Traces* is a need for dealing with imperative queries, when the user precisely points out to the types of services to be executed, just like in most of the considered medical scenarios. Moreover, *Traces* enable to associate the services types (and also their concrete instances) with attributes like price, duration, location or quality, without affecting the existing structure of the language.

The attributes of the class *Trace* are the following: *level*, *block*, *serviceType*, and *serviceName*. The first two of them aim at storing an information about a position of the service in the scenario generated, while the next two are used to identify the service executed. For example, if the service *GeriatricianSmith* is the first of the scenario, then the attributes of the trace *t* produced by this service are *t.level=0*, *t.block=0*, *t.serviceType="GeriatricianVisit"*, and *t.ServiceName="GeriatricianSmith"*.

Traces enable to express certain requirements on sequences of services, both on the level of service descriptions and while specifying users goals. For example, *SurgeonVisit* can require seeing a general practicioner earlier by including *x:Trace* in its *in* and *x.ServiceType="GPVisit"* in its *preCondition*. The descendants of *Trace* can introduce additional information. For example, a class *TimedTrace* with the attributes *start* and *stop* brings in time of the service execution. *PriceTrace* with the attribute *price* provides information about the service price, while *LocationTrace* with the attribute *location* introduces the information about the place where the service operates.

The user specifies its goal in the form of a *user query*, which defines what he "posseses" (the *initial world*) and what he "wants to posses" (the *effect world*), together with these of their features that are of his interest, using names of the classes from the branch rooted at *Object* and names of their attributes to this aim. For example, the user John Gold can specify that possessing "nothing", he wants to possess the object *p:Patient* with *p.First_name="John"* and *p.Last_name="Gold"*, which means that he wants to become a patient. The goals can be also specified in terms of traces (i.e., names of the classes from the branch rooted at *Trace*). This enables to express that the user wants the scenario generated to contain a service ofcertain type (e.g., by specifying that the effect world should contain *t1:Trace* such that *t1.ServiceType="SurgeonVisit"*) or a service of a concrete provider (e.g., by extending the above requirement by adding *t1.ServiceName="SurgeonSmith"*). Traces enable also to require a given ordering of services in a plan (by the use of the *level* attributes), or a given ordering of groups of servces (by the use of the attribute *block*) - for example, one can require the effect world to contain *t1,t2:Trace* such that *t1.ServiceType="GPVisit"*, *t2.ServiceType="SurgeonVisit"* and *t2.level<t1.level*, i.e., to see a GP after seeing a surgeon. Using other types of traces enables to influence the cost of services proposed, their time, location etc.

Our project follows the idea of separating two phases of the planning process. The first phase of searching for a sequence of services whose execution satisfies the user's goal is called the ***abstract planning***. It involves searching for sequences of **types** of services, which can transform the set of objects of the initial world into the set of objects of the effect world. The user's query is redefined to discard all the expressions involving concrete values of the attributes. For example, *p.Last_name="Gold"* is replaced by the requirement that the corresponding attribute is assigned a value - *isSet(p.Last_name)*. The abstract planning process is based on the bounded backward search algorithm, which starts from the final world and matches abstract services (special instances of service classes described before), which are capable to produce a desired set of objects (with the appropriate attributes set), building this way a graph whose nodes are sets of objects, and the edges are labelled with service types. This "preliminary" phase enables to limit the number of concrete (real-world) services considered while creating the final scenario as only these of appropriate types will be taken into account. Obviously, in the case of queries involving traces the role of the abstract planning phase is limited. The user can specify fragments of the abstract plan "by hand", using the appropriate attributes of traces. The next phase of the planning process, called the ***concrete planning***, aims at finding a sequence of **instances of service types** (concrete services) corresponding to an abstract plan obtained from the previous phase. Contrary to the abstract planning, this phase takes into account all the requirements specified in the query, i.e., also these involving concrete values of attributes. The planning process exploits an SMT-based model checking procedure, which is discussed in the next section.

3 Main Features and Implementation of Harmonics

Harmonics is a scheduling system that has been implemented for the Rehabilitation and Cosmetology Centre (CRiK) in Poland. The centre offers various types of medical services for its direct clients as well as for other medical facilities. The definitions of needs and possibilities of satisfying them are specified by a relatively complicated semantics. Additionally, availability of certain resources in many cases can be determined only dynamically, by querying external independent data sources. Before implementing Harmonics, due to the lack of IT solutions, the querying process was performed in an "unformalised" way, i.e., by phone or by e-mails. The knowledge obtained this way could not be processed automatically. The most important conclusions from the analysis of the functioning of CRiK, and from the users' expectations are as follows:

- The main goal of the system is to make the scheduling of treatments easier and more convenient, and also to automate some internal procedures,
- The most common case is to schedule a series of treatments w.r.t. patient preferences and resources restrictions,
- The single steps of the whole process can be realized by various service providers cooperating with CRiK,
- Some aspects of the abstract and concrete planning processes should be significantly adapted to meet the specific CRiK requirements.

The implementation of Harmonics, presented in the next part of this section, was designed to satisfy the above requirements.

The overall view of the Harmonics components is presented in Fig. 2. The ontology designed for CRiK was discussed in Sec. 2. The main software components of the system are as follows: the Repository, the Graphical User Interface (GUI), and the Planner. The aim of the Repository is to store information about the available services and their types (according to the ontology). Currently, it is implemented on the top of jUDDI - a popular UDDI implementation. GUI is a GWT web application that enables the user

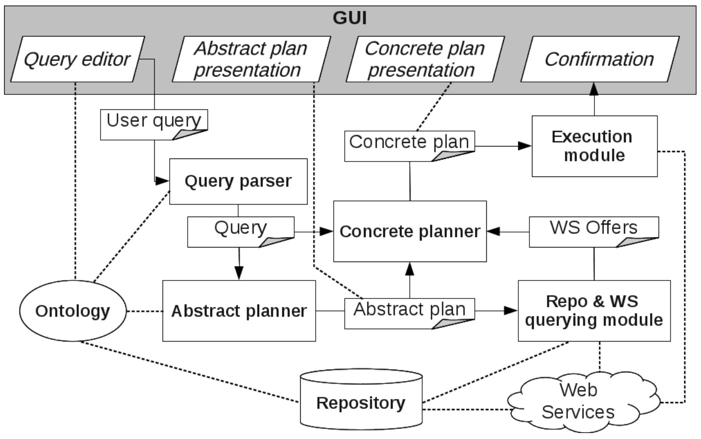
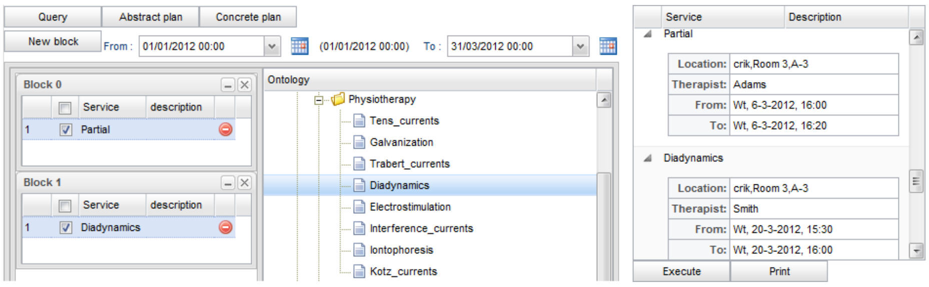


Fig. 2. The Harmonics overview



From: 01/01/2012 00:00 (01/01/2012 00:00) To: 31/03/2012 00:00

Block 0

	Service	description
1	Partial	

Block 1

	Service	description
1	Diadynamics	

Ontology
- Physiotherapy
  - Tens_currents
  - Galvanization
  - Trabert_currents
  - Diadynamics
  - Electrostimulation
  - Interference_currents
  - Iontophoresis
  - Kotz_currents

Service	Description
Partial	

Location:	crik,Room 3,A-3
Therapist:	Adams
From:	Wt, 6-3-2012, 16:00
To:	Wt, 6-3-2012, 16:20

Diadynamics

Location:	crik,Room 3,A-3
Therapist:	Smith
From:	Wt, 20-3-2012, 15:30
To:	Wt, 20-3-2012, 16:00

Fig. 3. On the left: the query editor, on the right: a fragment of a concrete plan

interaction with the system components. The Planner is a set of tools (represented by rectangles in the figure) for processing user queries (*Query parser*), creating plans (*Abstract and Concrete planners*), and interacting with the repository and with the web services (*Querying and Execution modules*). The rectangles with the right-bottom corner wrapped depicted in the figure correspond to the internal system objects. They are labels of the solid arrows which stand for a flow of objects. The dashed lines represent making use of some resource by a software component.

Let us now follow an example scenario, while giving more details concerning the implementation of individual components. First, using the *Query editor* (see Fig. 2 and Fig. 3), the user introduces a request, e.g., *I want to take a partial massage, once a week, for 10 weeks, and then a series of 5 diadynamics, every 2 days.* The user drags arbitrary services from the ontology tree at the right hand side and drops them to blocks of a plan at the left. The blocks are intended to enforce the order of an execution of the services. Each block of services is scheduled for execution when all of the services from the previous block have been completed. Putting some services in the same block means that they can be executed in an arbitrary order. Each of the services choosen can be parameterized by assigning a set of constraints, e.g., repeat conditions or specific requirements on the service date, time or location. The user should also specify an acceptable timing interval, providing the earliest start date and the latest end date of the whole sequence of services. The editor enables to hide the query language from the user offering a friendly and intuitive interface instead. The query of a formal syntax is produced in an automated way. For example, considering time interval from January, the 1st to the end of March of the current year, the formal query is as follows: *FROM null WHERE null TO repeat(t0:TreatmentTrace, 10, every 1 weeks), repeat(t1:TreatmentTrace, 5, every 2 days) WHERE _globalStart="2012-01-01 00:00" and _globalStop="2012-03-31 23:59" and t0.serviceType = "PartialMassage" and t0.block = 0 and t1.serviceType = "Diadynamics" and t1.block = 1.* As it is easy to see, the requirements on the service types and their ordering are expressed in terms of traces.

It should be mentioned also that the *repeat* statement is one of the novelties (comparing with [7, 8]) introduced to respond to the specificity of the domain, where the common case is to repeat some kind of treatment a number of times. Optionally, the repeat period can be given, just like in the example above. This construct makes editing of the query easier, as the user does not need to choose a service several times if he wants to repeat it. The user query is then processed by the *Query parser*, transformed to the internal representation, and made available to the *Abstract planner* and the *Concrete planner* (see Fig. 2).

The *Abstract planner* uses the knowledge from the OWL ontology and the query (rebuilt by discarding the concrete values as described before) for generating abstract plans, which are visualized and presented as sequences of service types. The user is asked to choose one of them to be concretized. Due to the fact that specificity of the area implies the queries to have a more imperative nature than in a typical case (the users typically enumerate directly the services they want to use) the role of the abstract planning is not so fundamental. However, using the knowledge from the ontology can introduce to the plan services not required directly by the user. In our example, the abstract planner returns the sequence of service types: *Registration*, 10 occurences of *PartialMassage*, and 5 occurences of *Diadynamics*. The *Registration* service, although not required directly by the user, is necessary in the plan as it "produces" a *Patient*, required by all the treatment services but not existing in the initial world.

Next, basing on the abstract plan and the user query, the *Repo & WS querying module* (RQM for short) examines the repository for the registered web services realising the types of services from the abstract plan. In our example the repository will be asked: "Give addresses of all the services of the type *PartialMassage*, and of the type *Diadynamics*". After getting an answer the RQM queries for offers the web services obtained (where by an offer we mean a service's declaration to execute under certain conditions). In our example the services will be asked: "Give the dates and time, between 2012-01-01 00:00 and 2012-03-31 23:59, when the treatment procedure can be performed" (the query contains no other constraints than these on the time period to be considered).

The next step is to run the *Concrete planner*. Its input are as follows: the (original) query, the abstract plan choosen to be concretised, and the offers collected for this plan (a single offer corresponds to a possible realisation of a single step of the plan, i.e., executing one service of a given type). It is possible to run this planning using one of the two methods. The first one, inherited from PlanICS [8], is based on a satisfiability checking (SAT). The new one is realized by a translation to an instance of the SMT [2] problem. An SMT-solver checks satisfiability of the formula which is the conjunction of the disjunctions representing particular offers, and an expression encoding the conditions specified in the query (e.g., repeat period constraints) and resulting from the abstract plan (e.g., the order of services). If this SMT instance is satisfiable, then a sequence of concrete services, whose execution satisfies the user's goal is decoded from the valuation returned by the solver. Going into more details, the attributes of the objects and the traces are encoded as SMT variables, and their values are mapped into natural numbers. For example, date-time values from our query are encoded as follows: the beginning of the considered period of time, the *_globalStart* value, is mapped to 0. All the date-time values are then related to the *_globalStart* value, according to a certain time scale. Currently the time scale is 5 minutes, which means that the value 10 represents the point in time 50 minutes after *_globalStart*. The SMT instance is encoded (using our original library) in SMT-LIB2 [4] format, which enables to use any compatibile SMT-solver. In the current version we make use of the Z3 [6] solver.

In the case of typical queries, involving from a few to several dozens of services, and from several hundreds to about 20000 offers, the total time of computations can

Offers	Time [s]		Mem [MB]	
	plain	interval	plain	interval
5866	3.06	4.67	115.07	117.19
10848	9.17	11.08	323.39	324.59
13241	15.27	17.92	475.80	477.68
17721	27.41	33.44	834.35	835.54

Offers	Time [s]		Mem [MB]	
	plain	interval	plain	interval
8281	2.38	5.18	90.13	91.31
12697	5.09	8.92	187.42	188.48
16561	8.33	19.60	289.37	291.95
21253	12.54	19.21	448.32	449.90

Table 1. Time and memory consumption of the concrete planner. On the left - the plan of depth 16 for the scenario: *registration, 10 massages, and then 5 diadynamics*, on the right - the plan of depth 24 for the scenario: *registration and 23 bioptrons*. The columns headed *interval* contain the results with additional constraints on the repeat frequency.

vary from a few seconds to about 30 seconds. The concrete planning phase seems to be the most time- and memory-consuming element. Table 1 displays some statistics of our SMT-based solution. The columns headed *interval* of the left table contain the results for the query being the working example of this section.

The concrete plan computed is visualized (see Fig. 3) and presented to the user. If the user accepts it, the *Execution module* invokes the services. Again, the specificity of the domain makes things simpler: an execution of a service consists in scheduling an appointment only, so no execution engine is necessary. Obviously, always something unexpected can happen. At the moment we follow the simple transactional policy: when any step of the plan could not be successfully executed, we cancel all of the already scheduled appointments, and the user can repeat either the WS querying and concrete planning phases, or the whole planning procedure.

4 Final Remarks

Harmonics is a specialized implementation of the concept which can be applied to various domains, enabling to build an integration system for distributed services of a common characteristic (e.g., transport, accomodation, reservation in time). More generally, a similar system can be implemented in every domain in which we have to plan an access to some resources with an independent management and optimize the plan by customized quality measures.

Comparing Harmonics to its ancestor Planics [8], we can point out to an easier and more natural handling of relations between services thanks to the concept of *Traces*. Another advantage appears in translating semantics from different IOPR [9] services ontology - it is simpler and more natural. On the other hand, a modular architecture of the system allows to take advantage of a new and more efficient planning solution based on SMT-solvers. The efficiency follows not only from applying the SMT-based technique, but also from the extended role of the querying module - the concrete planner deals now only with these of the parameters whose exact values cannot be determined by querying concrete services. Moreover, the planning mechanism related to the time have been improved. A further contribution of Harmonics is in an extended language of queries, enabling to express more requirements ocurring is practice. Its new elements are not only these which follow directly from introducing traces (like specifying requirements on ordering of services or their groups, or time or price of particular services), but also expressions enabling to require repetitions of services (the *repeat* statement) and summary constraints on the whole plan (e.g., *_globalStart*, *_globalStop*).

References

1. S. Ambroszkiewicz. *EnTish: An Approach to Service Description and Composition*. ISBN 83-910948-7-1, ICS PAS, Ordona 21, 01-237 Warsaw, 2003.
2. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Int. Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.
3. S. Bahadori, S. Kafi, K. Zamani far, and M. R. Khayyambashi. Optimal web service composition using hybrid GA-Tabu search. *Journal of Theoretical and Applied Information Technology*, 9(1):10–15, 2005.
4. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *Proc. of the 8th International Workshop on SMT*, 2010.
5. A. Blum and M. L. Furst. Fast planning through planning graph analysis. *Journal of Artificial Intelligence*, 90(1-2):281–300, 1997.
6. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.
7. D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Półrola, and M. Szreter. Web services composition - from ontology to plan by query. *Control & Cybernetics*, 40(2):315–336, 2011.
8. D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Półrola, M. Szreter, and A. Zbrzezny. PlanICS - a web service compositon toolset. *Fundamenta Informaticae*, 112(1):47–71, 2011.
9. A. Gómez-Pérez and J. Euzenat (Eds.). The semantic web: Research and applications. In *Proc. of the 2nd European Semantic Web Conference*, volume 3532 of *LNCS*. Springer, 2005.
10. S. Hoelldobler and H. P. Stoerr. Solving the entailment problem in the fluent calculus using binary decision diagrams. In *Proc. of the Workshop on Model Theoretic Approaches to Planning at AIPS2000*, pages 18–25, 2000.
11. J. Hoffmann, I. Weber, J. Scicluna, T. Kaczmarek, and A. Ankolekar. Combining scalability and expressivity in the automatic composition of semantic web services. In *Proc. of the 8th Int. Conf. on Web Engineering (ICWE'08)*, pages 98–107. IEEE Computer Society, 2008.
12. H. Kautz and B. Selman. Blackbox: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98*, 1998.
13. M. Klusch, A. Gerber, and M. Schmidt. Semantic web service composition planning with OWLS-XPlan. In *Proc. of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, pages 55–62. AAAI Press, 2005.
14. S. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for web service composition. In *Proc. of the 11st Int. World Wide Web Conference (WWW'02)*, 2002.
15. D. Roman, J. de Bruijn, A. Mocan, H. Lausen, J. Domingue, C. Bussler, and D. Fensel. WWW: WSMO, WSML, and WSMX in a nutshell. In *Proc. of the 1st Asian Semantic Web Conference (ASWC'06)*, volume 4185 of *LNCS*, pages 516–522. Springer-Verlag, 2006.
16. M. Sheshagiri, M. desJardins, and T. A. Finin. A planner for composing service described in DAML-S. In *Proc. of Workshop on Planning for Web Services, Int. Conf. on Automated Planning and Scheduling*, pages 28–35, 2003.
17. T. Vitvar, A. Mocan, M. Kerrigan, M. Zaremba, M. Zaremba, M. Moran, E. Cimpian, T. Haselwanter, and D. Fensel. Semantically-enabled service oriented architecture : concepts, technology and application. *Service Oriented Computing and Applications*, 1:129–154, 2007.