Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 248 (2009) 161–171
www.elsevier.com/locate/entcs

Dinapter: Automatic Adapter Specification for Software Composition

José Antonio Martín[1] Ernesto Pimentel[2]

Universidad de Málaga, Spain

Abstract

Enterprise systems rely heavily on compositional software like Software Components or Web Services. The composition of this software allows software reusability, greater productivity and reduced costs. However, these components are black-boxes and their direct reuse is prevented by incompatibilities between their interfaces. There are several approaches focused on Software Adaptation which are capable of solving incompatibilities at signature and behavioral levels, but these approaches require abstract specifications which specify how the incompatibilities can be resolved. The generation of these specifications is an open issue and specifications are normally handmade, which forces the designer to understand the subtleties of the components. In this paper we present Dinapter, a tool that automatically generates specifications being given the component behavioral descriptions written in abstract BPEL. Dinapter complements the aforementioned approaches and allows the automatic adaptation of compositional software.

Keywords: software adaptation, adapter specification, interoperability.

1 Introduction

Software Components and Web Services[3] are widely used in enterprise systems to allow software reusability, greater productivity and reduced costs. However, the composition of this software is not an easy task because the system components are usually designed in different contexts and therefore they present incompatibilities that require adaptation. Most of the time, components cannot be reused as they are, because interactions among them would lead to an erroneous execution, namely a mismatch. In practice, incompatibilities may be caused by message names which do not correspond (components interact on the same message names), or when the order of messages which is not respected, if a message in one component which

[1] Email: jamartin@lcc.uma.es

[2] Email: pimentel@lcc.uma.es

[3] In the sequel, we use *component* as general term covering both software components and services, i.e., a software entity to be composed within a system.

doi:10.1016/j.entcs.2009.07.066

has no counterpart, or mismatches between the arguments of the messages. These incompatibilities lead the whole system into deadlock states.

Bracciali *et al.* [4] developed a formal methodology to automatically derive adapters from component interfaces which include their syntactic and behavioral descriptions. These adapters are capable of solving incompatibilities by intercepting the communication between the components. However, this methodology requires abstract adapter specifications that contain a mapping between the actions of the components in such a way that when the adapter applies these correspondences, all the components cooperate properly and end up in a final state. The design of these specifications is a manual error-prone task which obliges the designer to have a full understanding of all the component details.

In this work we present Dinapter [4], a tool for the automatic generation of specifications for *behavioral adapters* (adapters which overcome incompatibilities at signature and behavioral levels). Dinapter accepts as input the behavioral interfaces of the components written in abstract *BPEL* [1] and it returns adapter specifications which describe how the mismatches (in signature and behavior) can be resolved. These specifications are used by other proposals [7,11] to generate the final adapter.

In the following section (Section 2) we will present an example to illustrate how Dinapter works. In Section 3 we will explain the parameters of our tool along with several results obtained. We will comment on related work in Section 4. Finally, we will present future work and some conclusions in Section 5.

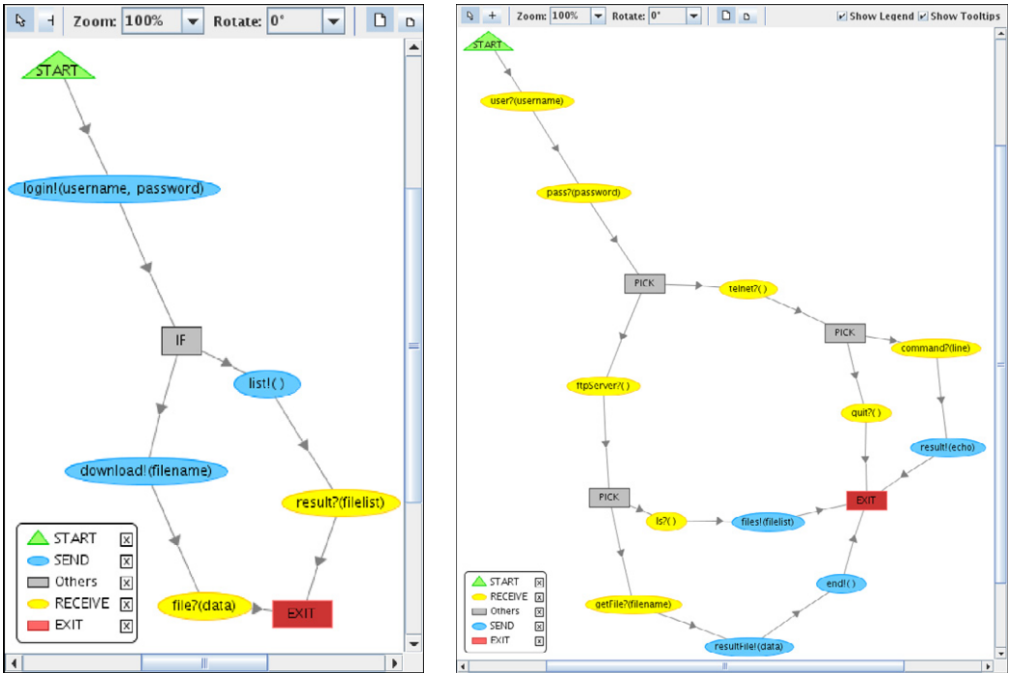
2 A Motivating Example

In Figure 1 we present two incompatible components. Emissions (`!`) and receptions (`?`) have their arguments in parentheses. The component on the left-hand side (Figure 1(a)) is a simple FTP *client* which connects to a service using its credentials (`login!(`*`username`*`,`*`password`*`)`) and it can request a file (`download!(`*`filename`*`)`) or the list of available files (`list!()`). The client receives the reply for its requests (`file?(`*`data`*`)` and `result?(`*`filelist`*`)`, respectively) and it ends. Listing 1 contains the abstract BPEL description of the client [5].

On the other side, the *server* (Figure 1(b)) requires authentication but, unlike the client, this authentication is split into two messages: `user?(`*`username`*`)` and `pass?(`*`password`*`)`. The server supports telnet and FTP sessions on request (`telnet?()` and `ftpServer?()`, respectively) but we can ignore the telnet functionality as long as the adapter requests the FTP service, which is the behavior expected by the client. The server presents name mismatches in every message but it has similar actions for file and list requests (`getFile?(`*`filename`*`)` and `ls?()`), and their responses (`resultFile!(`*`data`*`)` and `files!(`*`filelist`*`)`). There are two more things to notice: (i) the server emits an exit notification when a download ends (`end!()`); and (ii) the server has an action (`result!(`*`echo`*`)`) with the same name as the client's `result?(`*`filelist`*`)`, but these actions have different arguments and

[4] Available at http://sourceforge.net/projects/dinapter.

[5] The abstract BPEL description of the server is omitted because of space limitations.



(a) The client behavior.

(b) The server behavior.

Figure 1. Behaviors of the components.

semantics.

```
Archivo  Editar  Ver  Terminal  Solapas  Ayuda
42664   [18:54:33,249] INFO  solutions -  *** 4 out of 4 solutions found ***
----------------- Specification #0 ----------------------
; ◊ end!( )
download!(filename); file?(data) ◊ ftpServer?( ); getFile?(filename); resultFile!(data)
list!( ) ◊ ftpServer?( ); ls?( )
login!(username, password) ◊ user?(username); pass?(password)
result?(filelist) ◊ files!(filelist)
----------------- Specification #1 ----------------------
; ◊ end!( )
download!(filename); file?(data) ◊ ftpServer?( ); getFile?(filename); resultFile!(data)
list!( ); result?(filelist) ◊ ftpServer?( ); ls?( ); files!(filelist)
login!(username, password) ◊ user?(username); pass?(password)
----------------- Specification #2 ----------------------
download!(filename); file?(data) ◊ ftpServer?( ); getFile?(filename); resultFile!(data); end!( )
list!( ) ◊ ftpServer?( ); ls?( )
login!(username, password) ◊ user?(username); pass?(password)
result?(filelist) ◊ files!(filelist)
----------------- Specification #3 ----------------------
download!(filename); file?(data) ◊ ftpServer?( ); getFile?(filename); resultFile!(data); end!( )
list!( ); result?(filelist) ◊ ftpServer?( ); ls?( ); files!(filelist)
login!(username, password) ◊ user?(username); pass?(password)
---------------------------------------------------------
Do you want to search for different solutions (y/N)?
```

Figure 2. Output of Dinapter with several specifications for the components in Figure 1.

There are several possible adapter specifications to overcome the mismatches between the components in the example (some are displayed in Figure 2). Every

Listing 1 Abstract BPEL description of the *client*.

```
<?xml version="1.0" encoding="utf-8"?>
<process name="client-which_pick">
    <!-- Declarations: partnerLinks, variables, ... -->
    <invoke operation="login">
        <toParts>
            <toPart part="username"/>
            <toPart part="password"/>
        </toParts>
    </invoke>
    <if><condition opaque="yes"/>
        <sequence>
            <invoke operation="download" inputVariable="filename"/>
            <receive operation="file" variable="data"/>
        </sequence>
    <else>
        <sequence>
            <invoke operation="list"/>
            <receive operation="result" variable="filelist"/>
        </sequence>
    </else></if>
</process>
```

line in a specification is a mapping between the emissions and receptions of the components separated by diamond shapes ($\diamond$). Informally[6], the adapter which complies with a specification must perform the actions of one side of its mappings when it receives the actions of the other side.

In particular, the specification number 3 in Figure 2 has three mappings. The third mapping takes care of the authentication procedure, where two receptions on the server side must correspond to a single emission in the client side. Once the authentication is done, as far as the client needs an FTP session, the other two mappings start by sending `ftpServer?()` to the server and they continue matching the `list!()` and `download!(`*`filename`*`)` requests with their respective replies. It should be observed that the `end!()` message sent by the server is included at the end of the first mapping and therefore it will be received and dropped (not forwarded to the client) by the adapter.

The other remaining specifications are equivalent to the previous one. They are equivalent because they are just combinations where the mappings are split in places where they do not alter the adapter behavior, e.g., it is the same to match the `list!()` request and its reply in the same mapping (second mapping of the third specification) as to match them in one mapping for the request and another for the reply (third and fifth mappings of the first specification) because those operations

[6] See [4] for a formal description of the specifications and how they are used to generate the final adapter.

are in sequence in the component behaviors and therefore the mappings will be triggered one after another.

Any of these specifications, which are the output of Dinapter, are used as input of the methodology described in [4] to automatically derive the final adapter.

3 Dinapter

Dinapter accepts as input the behavioral description of the two components to adapt written in *abstract BPEL* [1] and returns a set of adapter specifications. Internally, the tool uses a combination of an *A* algorithm* and an *expert system*. The heuristic function used by the A* algorithm and the expert system rules represent the adaptation policy used to generate the adapter specifications.

Dinapter requires the components' arguments to have been previously matched (i.e., have the same name). Arguments with the same datatype and semantics in one component must have the same name in the other. This requirement can be accomplished by replacing the argument names by an abstraction of their datatypes. Dinapter will make the best effort to match the actions of the components depending on their arguments, types (emissions/receptions), and their position in the behavioral graphs.

When Dinapter starts, it loads the behavioral description of the components and it optionally displays their behavioral graphs (Figure 1). The expert system traverses the component behaviors generating partial specifications, it calculates the heuristic of those specifications, and it gives them to the A* algorithm which decides which partial specification must be continued.

The heuristic function (described in [10]) imposes an order among the generated specifications, placing first those specifications which best overcome the incompatibilities between the components. Therefore, it serves a double purpose as it guides the generation process (alleviating the explosion of partial specifications) and it returns first the specifications which best comply with the requirements expressed by the heuristic function and the expert system rules.

Eventually, it will find one or more (if there are several with the same heuristic value) valid specifications which pass the tests within the expert system, and it will return them (Figure 2). On request, the tool can discard the solutions found and perform another search iteration to find different specifications.

Table 1 contains some data gathered from Dinapter running several examples. The columns are as follows: the first (L) is the presence of loops in the example while *Pi* and *If* are the number of event driven conditions (`PICK`s) and regular conditional behavior (`IF`) in the components. The number of actions in the client and the server are *Cli* and *Ser* respectively. The next column (M) is the number of specification lines (or mappings) generated, T is the amount of search trees (conditions require several search trees), S is the number of generated specifications and ET and ES are the number of search trees and specifications explored before reaching a solution. U is the number of solutions that, in spite of being deadlock-free, adapt a branch of an event driven condition where no useful results are obtained from the adaptation, e.g.

Example	*L*	*Pi*	*If*	*Cli*	*Ser*	*M*	*T*	*ET*	*S*	*ES*	*U*	*Sols.*
e001		1	0	3	4	21	1	1	31	10	0	1
vod-3		2	0	5	6	52	1	1	120	32	0	2
e006		2	1	5	4	34	19	9	82	38	0	2
e010		2	0	9	7	76	1	1	191	43	0	1
e021	√	1	1	6	5	45	23	16	120	61	0	1
e007		0	1	2	7	45	50	18	180	100	0	9
e012	√	2	2	6	6	31	74	48	206	142	2	0(1)
e004b[7]		3	1	5	12	104	68	32	373	155	0	4
e018d	√	2	1	5	9	66	183	45	366	189	2	12
e002	√	1	1	3	6	48	149	53	412	215	2	4
e016		3	3	7	7	87	310	116	681	365	0	1

Table 1
Some results obtained from Dinapter.

a client which only connects and disconnects without doing any computation. This happens because the heuristic function and the expert system consider that it is better to connect and disconnect than to deal with the incompatibilities that would be found otherwise. Finally, *Sols.* is the number of valid solutions found. Some solutions can be equivalent to each other because they merge some specification lines into a single one (see Figure 2 for an example).

From this table it can be observed that the most relevant factor for the complexity of our tool is the number of local decision points which alter the execution flow (`IFs` and loops). These decision points have a bigger impact in the performance than the number of actions ($Cli + Ser$). Furthermore, if the adaptation policy is not appropriate for the problem (or the components have unsolvable incompatibilities), it may yield useless (U) results as in the examples e002, e012 and e018d. Nonetheless, if we execute another iteration of the process in the example e012, it returns a valid solution.

Another interesting point is the relevant role played by the A* algorithm and the underlying heuristic function which, even though there is a state explosion if the components are complex enough, it only needs to explore approximately half of the generated specifications before finding a proper adapter specification.

Dinapter has been successfully included in ITACA (*Integrated Toolbox for Automatic Composition and Adaptation*). This integration has enabled Dinapter to accept more input formats (*Microsoft Workflows* and *Symbolic Transition Systems*). Furthermore, the tool ACIDE (Figure 3, also included in ITACA) provides a graphi-

[7] e004b is the example given in Section 2.

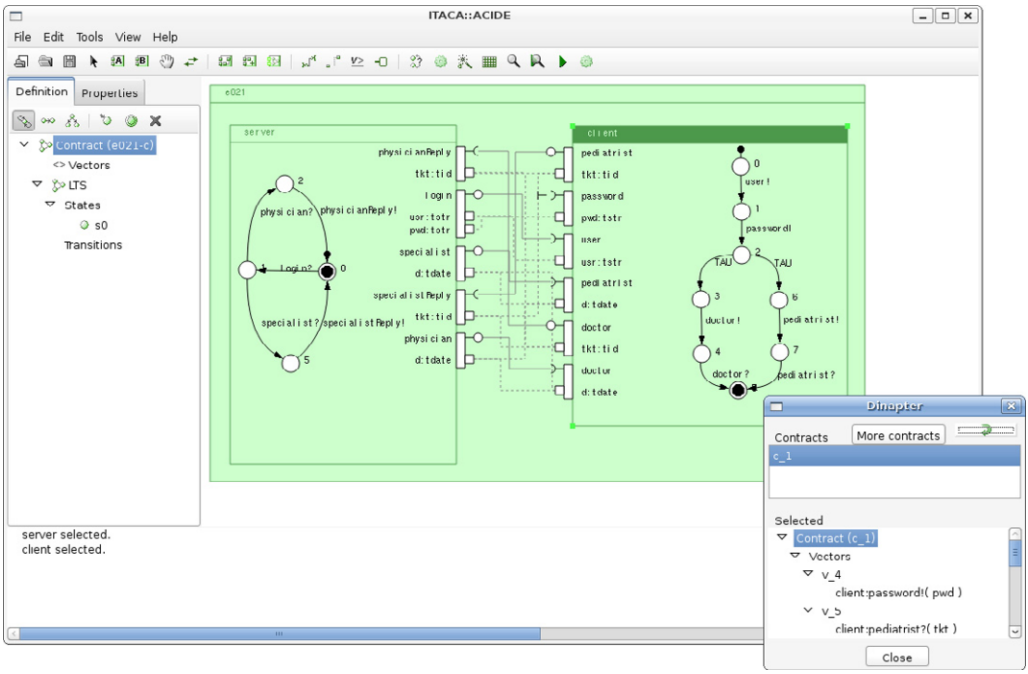


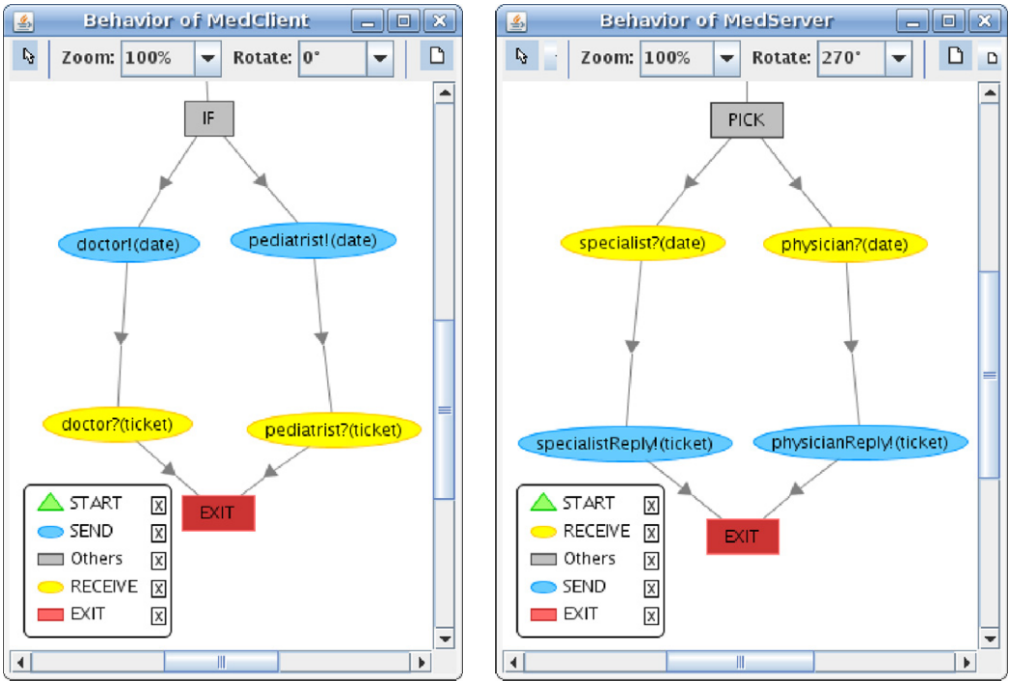
Figure 3. Integration between ACIDE and Dinapter.

cal interface where the developer can be assisted to design his own specifications or review, modify and accept the specifications generated by Dinapter.

Another tool in ITACA, called Compositor [11], accepts as input the specifications returned by Dinapter and applies the mappings in the proper order to generate the protocol of the final adapter.

The integration within ITACA has enhanced Dinapter by providing it with more information about the similarities between the components to adapt. A comparison of the semantics behind the operations to adapt is provided by another tool (Sim) which uses WordNet::Similarity [14]. The inclusion of this semantic information in Dinapter enhances the matching of name-mismatch situations, improves the adaptation of event driven conditions (PICKs) by taking into account the underlying semantics, and it reduces the number of search steps needed to find a correct specification.

For instance, Figure 4 shows part of the example e021. This example consists of a medical service (Figure 4(b)) where patients request dates with either a `physician` (P) or a `specialist` (S). On the other side (Figure 4(a)), the client component presents name mismatches and it requests `doctors` (D) or `pediatrists` (E). The arguments match and there is a reception for every emission so, leaving apart equivalent specifications and ignoring the names of the operations, the four combinations of branches are possible (D$\diamondsuit$P and E$\diamondsuit$S; D$\diamondsuit$P and E$\diamondsuit$P; D$\diamondsuit$S and E$\diamondsuit$S; D$\diamondsuit$S and E$\diamondsuit$P). However, through queries to WordNet, Dinapter recognizes that "physician" is a synonym of "doctor", "pediatrist" is an hyponym of "specialist", and it returns the right specification (D$\diamondsuit$P and E$\diamondsuit$S).



(a) The client behavior.

(b) The medical service behavior.

Figure 4. Adaptation with semantic information using WordNet::Similarity.

4 Related Work

Software adaptation is a very promising topic and it has been successfully applied to different implementation platforms such as BPEL [6] or Windows Workflow Foundation [9]. Several proposals [5,7,16] already focused on signature and behavioral adaptation. However, all these approaches require a manual specification of the adapter which may be tricky when component protocols are complicated. Our solution complements these proposals by generating adapter specifications from behavioral descriptions of components, which makes the adaptation process completely automated.

Moser *et al.* [12] developed a platform (VieDAME) based on ActiveBPEL for the monitoring and service adaptation of BPEL processes. They dynamically replace services based on QoS in a non-intrusive manner using aspect oriented programming. They use *Transformers* for service adaptation but these transformers must be designed manually. Their work can be complemented by our tool by automatically generating these transformers.

As regards automatic generation of adaptation specification, Schmidt and Reussner [15] focused on the synchronization of two components accessing, or being accessed, by a third one. They introduced an algorithm based on synchronous product computation to solve missing message incompatibilities, but their approach fails to overcome signature mismatches and behavioral incompatibilities like missing messages or message splitting / merging. Autili *et al.* [3] proposed a methodology for

the automatic synthesis of adapters considering as input behavioral descriptions of components and a specification of the interactions that must be enforced in the system. Then, their tool (Synthesis) generates composition code that exhibits only the specified interactions, and prunes those which lead to deadlocks. Similarly to [15], Synthesis does not overcome name mismatches, and some behavioral incompatibilities cannot be solved, such as message splitting / merging. In addition, their tool relies on a high-level description of the composition goal, and therefore does not work without such description.

Let us now mention two related works [6,13] that tackled Web services adaptation. In the first one, Brogi and Popescu [6] outline a methodology for the automated generation of adapters capable of solving behavioral mismatches between BPEL processes. In their adaptation methodology they use the *YAWL workflow* as an intermediate language. Once the adapter workflow is generated, they use lock analysis techniques to check if a full adapter has been generated or only a partial one (some interaction scenarios cannot be resolved). They solve message reordering incompatibilities but their approach fails with signature mismatches. In addition, even if we applied our approach to BPEL services as well, our approach is able to work with abstract descriptions of components/services that can be extracted from abstract BPEL but, due to its integration within ITACA, it also accepts other languages and platforms like *Symbolic Transition Systems* and *Windows Workflows*.

Motahari Nezhad *et al.* [13] presented a schema matching tool called COMA++ [2] for assisting the developer to adapt new versions of existing Web services based on the services WSDL signatures. Dinapter has some similarities with their work (the heuristic used by our tool plays a similar role to their *evidences*) and they introduce some interesting ideas about deadlock handling. However, although they are able to generate a mismatch tree that gather all protocol mismatches, its resolution is not automatic.

5 Concluding Remarks

In this work, we have presented a tool for the automatic generation of adapter specifications which overcomes signature and behavioral mismatches. The generated specifications successfully solve missing messages and they are able to merge and split messages depending on their arguments. There are several works in the literature [4,8,16] which use these specifications to automatically build behavioral adapters. Traditionally, these specifications were manually written and they required the designer to fully understand the details of the components involved. Our tool complements this previous work by automatically generating these specifications.

Dinapter achieves great versatility because, unlike other tools, it does not require any additional information (such as ontologies, use cases or sequence diagrams) to guide the specification process apart from the behavior of the components, and this is automatically extracted from abstract BPEL, Microsoft Workflows or Symbolic Transition Systems. Moreover, Dinapter can be used not only to adapt two compo-

nents but also to support service replaceability, generating specifications between the new service and the complementary behavior of the service to be replaced.

The combination of an informed-search algorithm and an expert system quickly solves simple mismatches but, the bigger the incompatibilities are, the more time is consumed exploring other ways to overcome them. This allows our tool to tackle different degrees of incompatibility but it wastes too much time when the incompatibilities are irremediable. One solution would be to complement our tool with an algorithm to automatically recognize these irremediable incompatibilities or an algorithm to cut component behaviors into smaller adaptable pieces.

Dinapter tackles behavioral and syntactic mismatches using a heuristic function, so it can still be misinformed by deceptive components where the behavior, arguments and operation names guide the generation process to a deadlock-free, yet invalid specification. However, we are currently working on including Linear Temporal Logic (LTL) formulas in the expert system. In this way, Dinapter will give us the option to further refine the specifications using both customized expert system rules and LTL properties.

Acknowledgement

This project has been partially funded by the Spanish Ministry of Science and Education (project TIN2008-05932) and Junta de Andalucía (project P06-TIC-02250).

References

[1] Alves, A., A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guíza, N. Kartha, C. Kevin, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri and A. Yui, editors, *Web Services Business Process Execution Language Version 2.0*, OASIS Standard, BEA Systems, IBM Corporation, Microsoft Corporation, SAP AG, Siebel Systems (2007), available via: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[2] Aumueller, D., H. Do, S. Massmann and E. Rahm, *Schema and Ontology Matching with COMA++*, in: *Proc. of SIGMOD'05* (2005), pp. 906–908.

[3] Autili, M., P. Inverardi, A. Navarra and M. Tivoli, *SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-Based Systems*, in: *Proc. of ICSE'07* (2007), pp. 784–787.

[4] Bracciali, A., A. Brogi and C. Canal, *A Formal Approach to Component Adaptation*, The Journal of Systems & Software **74** (2005), pp. 45–54.

[5] Brogi, A., C. Canal and E. Pimentel, *On the Semantics of Software Adaptation*, Science of Computer Programming **61** (2006), pp. 136–151.

[6] Brogi, A. and R. Popescu, *Automated Generation of BPEL Adapters*, in: *Proc. of ICSOC'06*, LNCS **4294** (2006), pp. 27–39.

[7] Canal, C., P. Poizat and G. Salaun, *Synchronizing Behavioural Mismatch in Software Composition*, in: *Proc. of FMOODS'06*, LNCS **6** (2006), pp. 63–77.

[8] Canal, C., P. Poizat and G. Salaun, *Model-Based Adaptation of Behavioural Mismatching Components*, IEEE Trans. on Softw. Eng. **34** (2008), pp. 546–563.

[9] Cubo, J., G. Salaün, C. Canal, E. Pimentel and P. Poizat, *A Model-Based Approach to the Verification and Adaptation of WF/.NET Components*, in: *Proc of FACS'07*, ENTCS **215**, 2008, pp. 39 – 55.

[10] Martín, J. and E. Pimentel, *Automatic Generation of Adaptation Contracts*, in: *Proc. of FOCLASA'08*, ENTCS, to appear.

[11] Mateescu, R., P. Poizat and G. Salaün, *Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques*, in: *Proc. of ICSOC'08*, LNCS **5364** (2008), pp. 84–99.

[12] Moser, O., F. Rosenberg and S. Dustdar, *Non-Intrusive Monitoring and Service Adaptation for WS-BPEL*, in: *Proc. of WWW'08* (2008), pp. 815–824.

[13] Motahari Nezhad, H. R., B. Benatallah, A. Martens, F. Curbera and F. Casati, *Semi-Automated Adaptation of Service Interactions*, in: *Proc. of WWW'07* (2007), pp. 993–1002.

[14] Pedersen, T., S. Patwardhan and J. Michelizzi, *Word-Net::Similarity - Measuring the Relatedness of Concepts*, in: *Proc. of Intelligent Systems Demonstrations, held in AAAI* (2004), pp. 267–270.

[15] Schmidt, H. and R. Reussner, *Generating Adapters for Concurrent Component Protocol Synchronisation*, in: *Proc. of FMOODS'02* (2002), pp. 213–229.

[16] Yellin, D. and R. Strom, *Protocol Specifications and Component Adaptors*, ACM Trans. Program. Lang. Syst. **19** (1997), pp. 292–333.