Merging overlapping orchestrations: an application to the Bronze Standard medical application

Clémentine Nemo, Tristan Glatard, Mireille Blay-Fornarino, Johan Montagnat

Merging overlapping orchestrations: an application to the Bronze Standard medical application*

Clémentine Nemo-Cailliau[1], Tristan Glatard[1,2], Mireille Blay-Fornarino[1], Johan Montagnat[1]
[1] UNSA-CNRS, I3S laboratory, RAINBOW team, Sophia Antipolis, France
[2] INRIA Sophia-Antipolis, Asclepios project, France

Abstract

Merging orchestrations is a crucial issue in the development process of service-based applications. However, merging orchestrations with overlaps is a manual and tedious process today. In this paper, we present a case-study on the Bronze-Standard, a medical imaging application built from Web-Service based orchestrations. We introduce the OMSM, an orchestration model supporting merging, that we designed to assist this process. Through a detailed analysis of the use-case, we show how our model helps the developer to obtain a proper composition of the application. There is still room for generalizing the approach to a broader set of orchestrations as discussed.

1. Introduction

Workflows have for long been developed in the context of the eBusiness community to provide a flexible and generic framework for compound tasks description and execution [1]. More recently, the scientific community has defined its own workflow languages and engines to take into account its needs for performance and data-intensive treatments [2, 3, 4]. The scientific workflows are often adopting a Service-Oriented Architecture (SOA) to achieve loose coupling among the services and to maximize code reuse. In particular, the Web-Services (WS) standard was massively adopted for designing SOAs [5].

Orchestrations are service-oriented workflows enabling the integration of different services, without giving up the loose coupling property [5]. Orchestrations are composed of *basic services*, such as Web-Services, and may also include other orchestrations, then called compound services [6]. When a compound service shares a basic service with other parts of the orchestration, those orchestrations are said *to overlap*.

Creating an orchestration is very demanding in terms of effort and domain knowledge. Different composition systems exist (Oracle BPEL Designer[1], the ADAPT framework [6]) but they rarely support compound services [7]. Moreover, composing overlapping orchestrations requires a particular *merging* procedure that detects shared variables and services and modifies the resulting orchestration accordingly, in particular to avoid redundant calls to services. None of the known composition approaches take it into account. For now when a developer has to compose overlapping orchestrations manually. She has to optimize the result: comparing the orchestration codes, checking the multiple service calls and merging the orchestration graphs. Even if some environments help the global understanding of the code by graphical notations [6, 8], the merge process cannot be automated, because the semantic of the services has to be understood.

Our goal is to automate the merge process as much as possible. We defend the idea that even if we cannot fully automatically deduce the merged orchestration, it is possible to merge some parts of the orchestrations and to identify potential conflicts for the developers in charge of resolving them based on their knowledge of the services semantics. In this paper, we propose an approach for merging overlapping orchestrations, by defining an *Orchestrations Model Supporting Merging* (OMSM) and guiding the developers with transfor-

*This work is partially funded by the AGIR French research program "ACI Masse de données" http://www.aci-agir.org.

[1] http://www.oracle.com/technology/bpel

mation rules to create new orchestrations. This model comes from a detailed analysis of the structure of orchestrations and is here applied to a particular medical imaging application. Our method is based on first order logic implemented by a set of Prolog rules.

Our work is illustrated on the Bronze Standard [9], a medical imaging application which involves the aggregation of several orchestrations as described in Section 2. Section 3 explains our merging algorithm and Section 4 details how the application is modeled. Finally, Section 5 explains the transformation rules.

2. The Bronze Standard application

The Bronze Standard application aims at statistically assessing the accuracy of various medical image registration algorithms in the absence of ground truth. Medical image registration is the process by which a geometrical transformation between two images acquired independently (usually a rigid transformation: translation and rotation) is estimated. The obtained transformations superimpose more or less precisely the 3D frame of the first image to the 3D frame of the second one. The inputs of a registration algorithm are a pair of images and a list of algorithm-specific parameters. The output is a rigid transformation (usually represented as a 4x4 transformation matrix). One important information about the Bronze Standard is that it uses multiple input image pairs and multiple registration algorithms to provide a statistical estimate. In particular, there is no fixed number of registration algorithms that can be used: the more algorithms available, the more accurate the procedure. The Bronze Standard orchestration needs to be recomposed, depending on the list of algorithms used. To ease this composition process, we have embedded all algorithms used in web services. More details on the Bronze Standard application can be found in [9].

2.1. Basic orchestrations

The Bronze Standard orchestration can be assembled from several basic orchestrations, each embedding a single registration algorithm. For the sake of clarity, we will only consider two registration algorithms in our example (namely `CM` and `PFM`) although much more could be envisaged as explained above. Figure 1 displays the two basic orchestrations (left and center) that have been designed by the developers using classical composition.

Each basic orchestration is made of an algorithmic part (upper part), in charge of the registration estimation, connected to an evaluation part (lower part), in charge of the accuracy evaluation. The algorithmic part includes the registration algorithm (`CM` on the left, `PFM` in the center). Both need a pre-processing step provided by the common `CL` service. `CL` service takes as input a pair of images and a numeric parameter. It returns 2 files (CrestLines) that are forwarded to the `CM` or `PFM` service. Further, the orchestrations differ: in addition to the two CrestLines files the `CM` registration algorithm takes as inputs the two images to register while the `PFM` registration algorithm takes as inputs an initial transformation parameter plus algorithm specific parameters (a string constant). The initial transformation matrix is an input for this basic orchestration but in the final Bronze-Standard orchestration, it needs to be computed by a former service, *e.g.* using the `CM` algorithm. Two files (`keys`) are produced by `PFM` and transferred to the `PFR` web service which also requires a numeric input parameter defined as a string constant. Both `CM` and `PFR` finally produce a transformation matrix and a comment string that are transferred to the common evaluation part of the orchestration.

The evaluation part of all the basic orchestrations is composed of 3 Web-Services sequentially linked (see bottom of figure 1). `Convert`, the first one, converts the transformation matrix resulting from the algorithmic part to a vector. This format conversion depends on the registration algorithm output. The vector is then concatenated to the input images names and to a comment string and written in a result file by the `Write` Web-Service. Once the whole data set has been processed by the `Write` Web-Service, the `Eval` one performs the evaluation of the accuracy. There is thus a coordination constraint between the `Write` and the `Eval` Web-Services. The inputs of the `Eval` service are the file where the results are stored and the name of the method to assess. This service finally produces the results of the accuracy evaluation.

The result of merging the `CM` and the `PFM` registration orchestrations is displayed on the right part of figure 1. The two input orchestrations have partial overlaps. The manual merging process is not trivial and

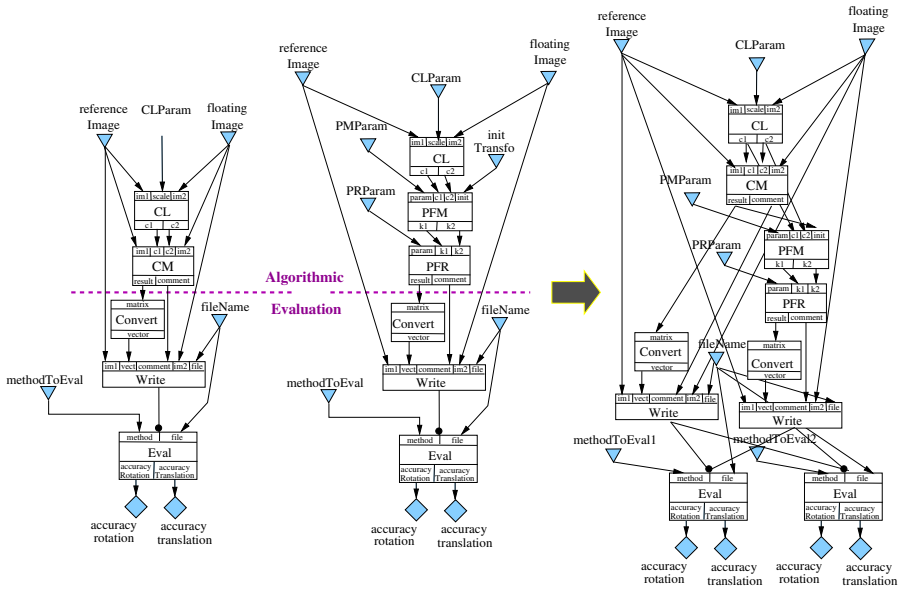

Figure 1. `CM` **(left),** `PFM` **(center) and merged (right) orchestrations.**

would become extremely tedious if considering many more registration algorithms. This application has a strong need for automated orchestration merging as it eases the inclusion of new algorithms that strengthen the evaluation results. However, some decisions depend on the semantics of the operation and can only be taken by the orchestration designer (*e.g.* using `CM` output as initialization parameter for `PFM`, invoking two `Eval` and two `Write` services independently, etc).

3. Semantic orchestrations merging

Currently, the known orchestration environments rely on the developers to perform orchestrations merging manually when overlapping orchestrations are involved. Merging orchestrations involve identifying overlaps, detecting conflicts, renaming variables, removing redundant calls, shifting order relationships between instructions, etc. This process conducted manually is tedious and error prone, especially when considering large size orchestrations. It generates a waste of human resources and, in case of repetitions due to changes in the input orchestrations, it could easily lead to errors and inconsistent results.

Alternatively, one could consider automated orchestration merging. However, full automation is infeasible unless semantic specifications of each service and data item are available. Without them we cannot deduce, for instance, whether the two `Write` and the `Eval` services should be merged or invoked independently in our example. To overcome this problem, we propose in this paper an orchestration merging process that *interacts* with the developer and (i) indicates potential unification points, (ii)automates the unification of service invocations or variables assignments in order to suppress potential sources of errors and (iii) automatically unifies input and output variables whose conceptual identity can be derived from previous unification steps.

3.1. Merging process overview

The three stages orchestration merging process is depicted in figure 2. First the input orchestrations are projected to the OMSM through the T_{in} transformation, thus making the process independent from the orchestration description format. The OMSM is then transformed iteratively (T_{m_k} transformations) to merge pairs of orchestrations as described below until a single merged result is produced. Finally, the result is back-projected into the desired description format through the T_{out} transformation.

The projection (T_{in}) and back-projection (T_{out}) make the OMSM independent from the description format used. In addition, the projection step enforces and checks properties that are required by the merging process. Indeed, the models of input orchestrations must satisfy the following properties:

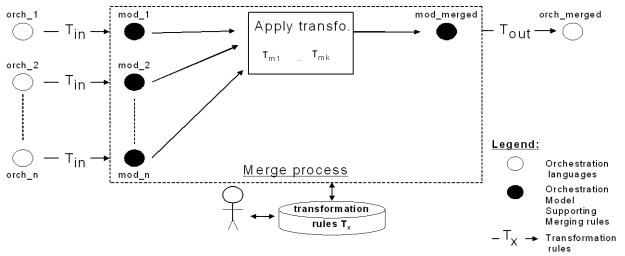

Figure 2. The interactive merge process.

- P_1 Input orchestration models have at most one invocation to a given basic service. To enforce it, multiple invocations to a same service are tagged as part of a structure named *complex invocation*.
- P_2 Input orchestrations have at most one *reply instruction* returning output data. If there are several outputs, they are encapsulated in a structure.
- P_3 Input orchestrations must satisfy a set of constraints such that they do not access concurrently in writing and reading their variables and there is no cycle in instructions invocation order. If this point is not valid for an orchestration, it is thrown out of the merge process.

The merging of two orchestrations verifying these properties will also verify them at each transformation step. Therefore, the OMSM transformation process can be reiterated on the resulting model. Moreover it respects the ordering of services given by the input orchestrations.

The OMSM transformations (T_{m_k}) are guiding the developer to achieve merging. The system is formalized in Prolog and the transformations follow specific Prolog rules. The potential unification points are detected (because they break directly or indirectly P_1 or P_2) and resolved according to their semantics given by the user. Invocations of a same service can either be unified in a single call or tagged to keep the multiple different calls. The transformation process automatically computes the resulting orchestration and updates the remaining potential unification points. After each transformation application, the P_3 property is checked. The process is iterated until obtaining a single orchestration respecting the properties P_1, P_2 and P_3 (with a single output and all redundant calls resolved).

3.2. OMSM representation

Orchestrations, such as the ones pictured in figure 1, are composed of *variables* (that can be read and assigned), *services* (that are invoked) and *dependencies* constraining the order of execution. Variables represent the input and output data of the orchestration services. They are characterized by a name and a type.

To deal with different kinds of service invocation, we introduce the concept of *instruction*. Each instruction is characterized by four parameters: an identifier, an ordered list of input variables, an output variable and an activity depending on the invocation. The service boxes depicted on figure 1 are decomposed in several instructions: to read inputs, invoke services, and write outputs. We consider the following activities:

- A *basic invocation* to a service represents a single service call. It may either be a unique service invocation or be part of a complex invocation (defined below). It is characterized by two parameters: a unique name (composed by the service name and the operation invoked) and the identifier of a complex invocation this basic invocation belongs to (null in case of unique invocation).
- A *complex invocation i* groups several basic invocations $i_1 \dots i_n$ to a same service. It is characterized by the name of the service and the set of basic invocations to this service. Complex invocations also define an invocation policy which details how the unification of basic invocations was resolved. Many different policies can be envisaged. In this paper we only use two of them that are depicted on figure 3. The Separate Policy (SP) corresponds to concurrent invocations of the services without input nor output unification. It corresponds to the case where a services invoked in two input orchestrations should be invoked twice independently in the merged orchestration. Similarly, the Synchronous Concurrent Execution Policy (SCEP) concurrently invokes the services without unifying inputs and outputs but it adds the following constraints: for every invocation i_k of the complex invocation i and for every j in the set of instructions of the orchestration such that j is executed before i_k we constrain every i_l ($l \neq k$) to be executed after j. This corresponds to the case where all predecessors (B and

C in figure 3) to the unified service (A) should be terminated before the multiple calls to the service can be triggered. Note that in other contexts, many more policies can be envisaged such as sequencing the unified service calls, etc.

- An *adapter* aims at writing or reading variables. We define it as a function which takes n input variables and returns a single output variable. Adapters are not subject to the same transformation rules as invocations.
- Finally, a *reply* activity finally corresponds to the orchestration outputs collection: results produced by an orchestration are returned through a special reply instruction.

In OMSM, we model an orchestration as a tuple and a list $o = ((i, V, I), \ell)$ where i is the identifier of the orchestration, V is the set of orchestration variables, I is the set of the orchestration instructions, and ℓ is the list of constraints on the instructions to determine the global partial order of execution.

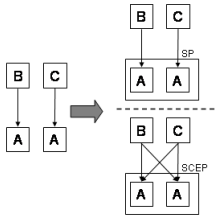

Figure 3. Policies for complex invocations

3.3. Transformation rules

Once input orchestrations are modeled in OMSM, they are transformed through the set of transformation rules described below. In practice, this process is implemented using the logic-based model in Prolog. The implementation is described with more details in Section 4. The transformation rules have different goals:

- *Unifying input variables*: all input variables are formalized in the orchestrations in conformity with OMSM. The programmer is asked to formulate relationships between input variables. Depending on the variable types, different relationships can be expressed. If the value of a variable is contained in another one, the variables are packed in a single structure and an adapter is added to access the sub-variable. Some input variables can disappear because their value is obtained from variables in another orchestration (see the *init transfo* variable in figure 1 for instance). At each step of the merging process the user can decide to assign or unify input variables. This operation involves to check that the variable is assigned before being used.
- *A single reply*: when an output variable is expected, there must not exist more than a single reply instruction. As in the previous rule, if we detect that there exists several reply instructions, the developer must specify relationships between them. Reply instructions are consequently merged.
- *Unifying invocations*: this rule detects a potential merging point when there are several basic invocations to the same service and when these invocations $(i_1 \dots i_n)$ are not part of a same complex invocation. The user can either unify them, or create a complex invocation. In the case of unified invocation, if the input variables differ an adapter is used to compose them. Output variables are unified automatically. A new basic invocation i is created that is subject to all the constraints on the unified instructions $(i_1 \dots i_n)$. In the complex invocation case, invocations $(i_1 \dots i_n)$ the user has to specify an invocation policy. Each basic invocation $(i_1 \dots i_n)$ references the new complex invocation i.
  According to this unification several adapter instructions can have the same input variables. When their output variables are only accessed in reading the adapter are automatically merged.

Those transformation rules can only reduce the number of variables and services invocations. Therefore, the merging algorithm terminates after a finite number of iterations. A formal proof of it is out of the scope of this paper.

4. Merging overlapping Scufl orchestrations

4.1. The Scufl description language

The Bronze-Standard orchestration is written with the Scufl language, from the myGrid UK eScience project [3]. This XML-based description language

is commonly used in the scientific community to describe orchestrations of services. It offers facilities to describe data-intensive applications such as ours. In particular, Scufl operators enable the description of several iterations of a service on a complete data set.

In Scufl, the basic elements of an orchestration are *processors*. Web-Services are a particular kind of processor, as well as string constants, local java classes and other ones. Web-Services are identified by a specific XML tag inside the *processor* one. This tag defines the WSDL document defining the Web-Service and specifies the operation to be invoked among the ones available in the Web-Service. Sources (inputs) and sinks (outputs) of the orchestration may also be defined. Sources allow the user to split the description of the orchestration from the one of the processed data. Data sets are instantiated by the user at run time and may contain several pieces of data, each of them resulting in an iteration of the orchestration.

Once the processors, sources and sinks of the orchestration are defined, links between them are expressed. There exists two different kinds of links: *data links* associate outputs of some processors with input of other ones depending on the produced data for their execution; *coordination links* specify a temporal dependency between the concerned services. They can be used to express data synchronization.

There is no convention about the graphical representation of Scufl orchestrations. In figure 1, we depicted sources and constants with blue triangles, sinks with blue diamonds and Web-Services with white rectangles. Input messages of Web-Services are represented by rectangles on top of the white boxes and output ones at the bottom. Parts of the messages are identified by sub-divisions of the message rectangles that are labeled to ease legibility. Data links are represented with arrows and coordination constraints with dot-terminated arrows (all links are data links expect the coordination link between services `Write` and `Eval`). To exploit our automated merging procedure, we have to define a transformation rule able to project Scufl orchestrations into OMSM.

4.2. Projecting on orchestration model

Transforming Scufl orchestrations to OMSM orchestrations consists in checking the P_3 property and in adapting resulting model to conform to properties P_1 and P_2. In the PFM orchestration example, we encapsulated the output variables `rotPFM` and `transPFM` in a new variable `resultPFM`.

Scufl enables data parallelism: multiple data sets can be pushed through the orchestrations inputs, resulting in the multiple execution of each service to process all input data segments (so called *iteration strategies* in Scufl). Currently, the OMSM only consider simple inputs. Supporting data parallelism will be the subject of future work as discussed in the conclusion.

4.3. Prolog representation

We transform Scufl orchestrations in orchestrations formulated in the Prolog logic language, conforming to the OMSM. Below, we give details of the orchestration instructions (`instr`) for the PFM orchestration displayed on figure 1.

The link between Scufl sources and sinks is described as follows: the input set of variables V is `[im0,im1,fN,method,transfo,cl,pm,pr]`. It represents the input set of variables `referenceImage`, `floatingImage`, `fileName`, `methodeToEval`, `initTransfo`, `CLParam`, `PMParam` and `PRParam` in figure 1. The set of instructions I is composed of 17 instructions `[i1...i17]` belonging to three types: `invoke` (service calls), `assign` (variable assignment) and `reply` (output collection). They correspond to basic invocations and adapters in the OMSM representation.

```
orch(pfm,[im0,im1,fN,method,transfo,cl,pm,pr],
    [i16, i17, i1, i2, i3, i4, i6, i5,
     i7, i8, i9, i10, i11, i12, i13, i14, i15]).

instr(i1,[im0,im1,cl],clOut,invoke(cl)).
instr(i2,[clOut],c1,assign(floatingCrest)).
instr(i3,[clOut],c2,assign(referenceCrest)).
instr(i4,[pm,c1,c2,transfo],pfmOut,invoke(pfm)).
instr(i5,[pfmOut],k1,assign(floatingKeys)).
instr(i6,[pfmOut],k2,assign(referenceKeys)).
instr(i7,[pr,k1,k2],pfrOut,invoke(pfr)).
instr(i8,[pfrOut],res,assign(transfo)).
instr(i9,[pfrOut],com,assign(comment)).
instr(i10,[res],convOut,invoke(convert)).
instr(i11,[convOut],vector,assign(resultVec)).
instr(i12,[im0,im1,com,vector],writeOut,
                                invoke(write)).
instr(i13,[fN,method],evalOut,invoke(eval)).
instr(i14,[evalOut],rotPFM,assign(rotation-rad)).
instr(i15,[evalOut],transPFM,assign(trans-mm)).
instr(i16,[rotPFM,transPFM],result,assign
                    (concat(rotPFM,transPFM))).
instr(i17,[result],void,reply).
```

The Scufl links enforce an invocation order between instructions. Below is the list ℓ of constraints in our model. Each is formalized in Prolog as `pred(instr1, instr2)`, enforcing the execution of the instruction `instr1` before the instruction `instr2`. The absence of constraints between two instructions formulates the concurrent execution. In the PFM orchestration case, the list of constraints is:

```
pred(i1, i2).pred(i1, i3).pred(i2, i4).pred(i3, i4).
pred(i4, i5).pred(i4, i6).pred(i5, i7).pred(i6, i7).
pred(i7, i9).pred(i7, i8).pred(i10, i11).
pred(i8, i10).pred(i11, i12).pred(i9, i12).
pred(i12, i13).pred(i13, i14).pred(i13, i15).
pred(i14,i16).pred(i15,i16).pred(i16, i17).
```

We apply transformation rules on this representation to guide the developer in the merging process.

5. Merging orchestrations

From the OMSM representation of the CM and PFM Scufl orchestrations, we can apply transformation rules[2] to compose the Bronze Standard. When potential unification points are detected the developers input is requested. Along the merge process, the `assign` instructions are automatically adapted and optimized. In our example, multiple reply instructions, multiple invocations to a same service (`CL`, `Convert`, `Write`, and `Eval`), and choice of input variables are the potential unification points detected.

Unifying input variables: Prolog rules applied to the list of input variables (`[im0,im1,pm,pr,fN, method,transfo,cl,pm,pr]` for PFR and `[im0,im1, s,fN,method,cl]` for CM) lead to conflicts. As the intention is to compare several algorithms on the same input images, the developer chooses to unify the pairs of images (`im0, im1`). The results have to be written in a single file. The the file name (`fN`) and CL constant (`cl`) are also unified by the user. Conversely, the `method` variable describing which method is evaluated needs to be different in both invocations. Two distinct variables are thus needed. The original variables are renamed into `method1` and `method2` to make the distinction. For the constants (`pm`) and (`pr`) existing only in the PFR, the developer simply preserves them in the input variables. Finally the developer unifies the `transfo` variable with the `result` output variable of the CM service. The `transfo` variable is not an input variable any more in the resulting orchestration.

Unifying invocations: based on the service names, the process detects the overlapping invocations and submits the remaining conflicts to the developer. First, the developer chooses to unify the two invocations of CL, because they correspond strictly to the same computation. Before operating the invocation unification, the variables in both orchestrations are renamed in order to avoid naming conflict: a variable named `x` in orchestration PFM (resp. CM) is renamed `xPFM` (resp. `xCM`), if it does not correspond to an input variable. The input variables of `CL` have already been unified. The process unifies the renamed output variables (`clOutCM`, `clOutPFM`) of `CL`. The developer chooses not to unify the overlapping invocations to `Convert` and `Write`. She selects complex invocation policies. In this case, she wants two concurrent executions and selects the Separate Policy. Consequently the process references the corresponding basic invocations and set their identifiers to the one of the complex invocations. For the two invocations of `Eval` the process suggests not to unify them because the (`method`) input variables have been distinguished. The developer chooses the Synchronous Concurrent Execution Policy to make the evaluation start only when both the `Write` invocations have been performed. Thus, the process increments the constraints list adding two dependencies between the `Eval` and `Write` invocations. This semantic conforms to the coordination constraint expressed in Scufl between these services.

Unifying reply instructions: The process detects multiple reply instructions and creates a list of output variables. The developer simply concatenates the pairs `returnPFM` and `returnCM` to have a single reply.

After applying each rule, the process checks the validity of P_3. In particular it checks that no deadlock and no concurrent access to variables have been introduced by the user during the merge process. If a violation of P_3 is detected, backtracking is possible and the user is requested to solve the problem.

6. Conclusion

We have addressed the problem of automatically merging several orchestrations into a single one. Be-

[2] For the sake of brevity, these rules could not be reproduced here but examples can be found in [10] or on `http://rainbow.polytech.unice.fr/adore` web site.

cause of the absence of semantic description of all the processes involved in the input orchestrations, the process cannot be automatic. Relying on the user knowledge allows us to cope with this problem. Our OMSM model uses a formal representation of orchestrations and a set of transformation rules to proceed with the automation. It is implemented in Prolog.

We have shown on a concrete example how the process is conducted and what are the benefit. The process not only eases the merging from a developer point of view but it also drastically reduces the risk of errors by automatically revealing all potential conflict points. To build our *Bronze Standard* medical imaging application, we merged two basic overlapping orchestrations that shared variables and invocations. Conflict points were resolved by suggesting to the user to apply transformation rules. This merging process has also been successfully applied to the problem of merging BPEL orchestrations in the context of a travel agency [10], which suggests that it generalizes.

OMSM was initially designed for control-flow workflow approaches. Data-flow languages such as Scufl exhibit different requirements that cannot be addressed by our approach. For instance, handling data-parallelism is not possible yet. A basic technique to cope with it would be to formulate data parallelism as a concurrent execution of the orchestration on different input variables. However, this is limited to the merging of orchestrations for input data sets whose size is known before the execution and does not take into account the dynamic behavior of service-based orchestration descriptions. For instance, merging basic orchestrations of the *Bronze Standard* application on clinical use cases for which the size of the image database is only known at runtime is not possible for now. Moreover, handling data-parallelism would imply to be able to represent iteration strategies, in order to define how a given service with multiple input ports is iterated on several data items. This could be managed by defining new policies for complex invocations.

Finally, another limitation is the intensive usage of interactivity made by the merging process. Each conflict point has to be solved by the user. We plan to overcome this limitation by the use of a knowledge base in order to capitalize on the user choices: the more the system would evolve, the more automatic it would become.

References

[1] W. M. Van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, July 2003.

[2] I. Taylor, M. Shields, I. Wang, and A. Harrison, "Visual Grid Workflow in Triana," *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 153 – 169, 2005.

[3] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: A tool for the composition and enactment of bioinformatics workflows," *Bioinformatics journal*, vol. 17, no. 20, pp. 3045–3054, 2004.

[4] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific Workflow Management and the Kepler System," *Concurrency and Computation: Practice & Experience*, vol. 18, no. 10, pp. 1039 – 1065, 2006.

[5] OASIS SOA Reference Model Technical Committee, M. MacKenzie, K. Laskey, F. McCabe, P. Brown, and R. Metz, "Reference Model for Service Oriented Architecture 1.0," OASIS, Tech. Rep., Oct. 2006.

[6] A. Bartoli, R. Jiminez-Peris, B. Kemme, C. Pautasso, S. Patarin, S. Wheater, and S. Woodman, "The ADAPT framework for adaptable and composable web services," *IEEE Distributed Systems Online*, vol. 6, no. 9, Sept. 2005.

[7] R. Khalaf, N. Mukhi, and S. Weerawarana, "Service-Oriented Composition in BPEL4WS," in *12th International World Wide Web Conference (WWW'2003)*. Budapest, Hungary: W3C, May 2003.

[8] S. Ben Mokhtar, N. Geogantas, and V. Issarny, "COCOA : Conversation-Based Service Composition for Pervasive Computing Environments," in *IEEE International Conference on Pervasive Services (ICPS)*, Lyon (France), 2006.

[9] T. Glatard, X. Pennec, and J. Montagnat, "Performance evaluation of grid-enabled registration algorithms using bronze-standards," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI'06)*, ser. LNCS, Copenhagen, Denmark, Oct. 2006.

[10] C. Nemo, M. Blay-Fornarino, G. Kniesel, and M. Riveill, "Semantic orchestrations merging - Towards Composition of Overlapping Orchestrations," in *9th International Conference on Enterprise Information Systems (ICEIS'2007)*, J. Filipe, Ed., Funchal, Madeira, June 2007.