# Tasks models merging for high-level component composition

Arnaud Lewandowski[1], Sophie Lepreux[2], and Grégory Bourguin[1]

[1]Laboratoire d'Informatique du Littoral (LIL)
50 rue Ferdinand Buisson, F-62100 Calais, France
{lewandowski,bourguin}@lil.univ-littoral.fr
[2]University of Valenciennnes, LAMIH
Le Mont-Houy, F-59313 Valenciennes Cedex 9
sophie.lepreux@univ-valenciennes.fr

**Abstract.** As users become more and more demanding about the software environments they use, they need environments offering them the possibility to integrate new tools in response to their emerging needs. However, most high-level component composition solutions remain out of reach for users. Thanks to an innovative approach that tends to provide more understandable components, we propose in this paper a new mechanism in order to assist high-level component composition. This approach proposes to realize this composition through tasks models assembling. The assistance we propose is based on an adaptation of tree algebra operators and is able to automatically merge tasks trees in order to assist high-level component integration in a more global environment.

## 1 Introduction

Many theoretical and empirical studies have already demonstrated the emerging nature of users' needs towards their activities and the environments supporting them [3,11]. Actually, many research works tend to integrate in the software environments the mechanisms suited to support these emerging needs, and to give the users the possibility to make these environments evolve. One solution is to allow the users to integrate tools inside their environment, i.e. to compose high-level software components. In order to be efficient and accurate, such integration should be fine and dynamic. Despite the great amount of work and advances that have been made in the field of component integration or composition, one must agree that the available solutions are still generally complex and always directed to a public of software development experts. The purpose of our work is to facilitate the fine and dynamic integration of tools (or high-level components) inside a global environment. In order to reach this goal, we can identify two aspects on which we have to work. Firstly, we should be able to provide more understandable components. And secondly, we should provide some automated or semi-automated assistance for composing such components. In this paper, we particularly focus on this latter aspect.

In the first part of the paper, however, we are going to introduce the Task Oriented (TO) approach that tends to provide more understandable components. This innovative approach proposes a way to construct high-level components that could be more easily integrated afterwards, especially thanks and through the use of tasks models. Following this approach, we propose to realize the integration of high-level components inside a global environment through the assembling of their individual tasks model in the more global task of the integrating environment. Even if interesting, this proposition raises some questions. We particularly focus in this paper on the means that could help or assist the realization of the merging between several tasks models. In the second part of the paper, we present a solution that has been developed during previous work about how to compose XML trees thanks to specific operators. Since the tasks models of our components are described in XML documents, we propose in the third part of the paper to adapt the XML tree composition solution to the merging of several tasks models. This proposition tends to assist the composition of high-level components that have been developed according to the TO approach. Finally, we illustrate this proposition with an example of high-level component composition through the assisted merging of their individual tasks models.

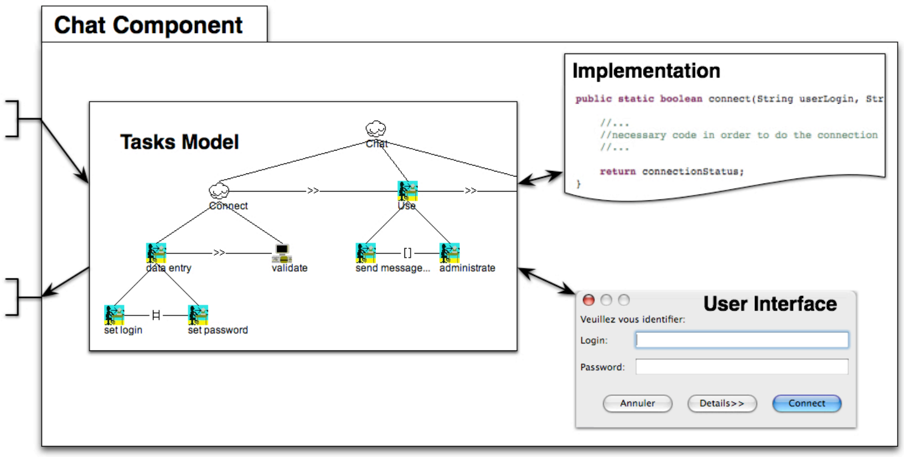
## 2 The Task-Oriented Design Approach

Software components composition is a large and complex research area. Besides, many technical solutions try to give it answers. For example, distributed components such as CORBA components [13], EJB (Enterprise JavaBeans) [1], or Web Services [4] have been conceived with the perspective of their future integration. Some of them are associated with composition languages [12] that allow the fine integration of these components or services inside software applications. One can notice that such technical solutions are exclusively usable by software development experts, especially because of their complexity, of their implementation cost and of the specificity of the used techniques [5]. However, these different methods follow the same principle: it is possible to dynamically discover objects on the Internet, to instantiate them, to discover their public methods and eventually their event channels, and to determine how the global environment will integrate and pilot them (through specific methods calls). Even if very useful, these mechanisms mainly bring a solution to the technical dimension of the problem. Indeed, the fact of finely and dynamically integrating a tool not only supposes that we are able to use it, but also that we understand *how* to use it. And even if some documentation supports — like the Javadoc for Javabeans components, or WSDL descriptions for Web services — may exist, this problem of semantic still remains. Every developer has been faced with this problem: introspecting the list of the public methods of a specific component, even with their documentation, is generally not sufficient in order to finely realize its integration: knowing the methods does not tell you — for instance — in which order you have to call them for this component to work properly. The Task Oriented approach is intended to give an answer to the semantic lack in high-level components and to raise the abstraction level of their composition.

Our approach is to consider that each high-level component aims at supporting a specific kind of activities. Our goal is to provide the means to contextualize the many tools or components involved in the realization of a global task. In other words, the global environment has to manage what we call the *inter-activities* [8], i.e. the links existing between the activities supported by the many tools integrated in a global environment in order to support a composed and global activity. We consider then that each component supports the task it has been designed for. Indeed, the designer of a specific tool or high-level component has created the underlying mechanisms and its interface in order to propose an adequate support for a specific generic task. Thus, a mailing component supports the realization of mailing tasks; a chat component supports synchronous discussion activities, etc. So we can consider that contextualizing a tool is equivalent to contextualizing an existing task into the frame of a more global task, such as *Co-writing an article for HCI-International*, where a mailing tool may be associated with a word processor, a chat and other tools. In order to facilitate this contextualization and to bring an answer to the dynamic integration problems, we propose to better use the component's tasks model, a kind of missing link that generally disappears between the design stage and the delivered code.

Actually, tasks models are generally used at the beginning of the software development process. But their use progressively fades during the process and finally disappears behind an object-oriented design approach inspired by the computer engineering background. This classical software design approach tends to transform tasks models into objects models, from which emerges implicitly the class-based structure of the produced component. The original tasks model is swamped, implicitly inscribed in the complexity of the produced source code. Indeed, task-oriented approaches are slightly used – or even not used at all – during the design and development cycle, namely after the requirements collection and analysis.

Nevertheless, at the stages where they are used, tasks models often serve, as shared objects, to help a better communication between the many actors (including the future users) implied in the complex software development process. Tasks models also contain useful information describing the functioning of the tool and serve a better understanding of it. The Task Oriented design approach tends to keep the benefits of tasks models during the whole software development process and even during composition or integration stages. In order to facilitate high-level component composition, the TO approach proposes to include the tasks model of a component within it [9]. This approach consists in the explicit preservation of the links between the functional source code and the tasks model it is based on. The Figure 1 summarizes part of the architecture of a high-level component developed according to the TO approach — or TO component. Not only the embedded tasks model adds some semantics to the component and should help in its understanding, but it could also be used for its integration. Thus, the developer of a specific component could specify on its tasks model which parts (or subtasks) could be *"shunted"*, i.e. realized by the global environment. For example, in the chat component of the Figure 1, the "connect" subtask could be shunted by the global environment calling the `connect()` method with the right arguments, which may have for effect to 'realize' the "validate" subtask, and as a consequence the "connect" subtask, skipping the corresponding interface. The purpose of this paper is not to describe in details how these links between the functional code and the tasks model are kept inside the TO component. Briefly said, the tasks model, described in a XML document, contains some information about the tasks that can be realized through the call of a specific method by any global environment integrating this component. For such tasks, a specific field indicates which method it corresponds to. These specific methods are grouped together into a kind of wrapper class so that the global integrating environment can do the appropriate calls to pilot the component. Such TO components are then developed with their future integration in mind, since the designers/developers are able to put in them the appropriate methods that will shunt some of their subtasks and then adapt their behavior.



```
public static boolean connect(String userLogin, Str

    //...
    //necessary code in order to do the connection
    //...

    return connectionStatus;
}
```

Veuillez vous identifier:
Login:
Password:

**Fig. 1.** Architecture of a TO component

Therefore, high-level component composition could be realized through the composition, or merging, of several tasks models in a more global one, supposing that these components are "TO components" that include their tasks model. The global environment, managing the global task, could integrate the many tools by integrating their tasks models. One of the benefits of this approach is that it raises the abstraction level required for assembling components. It removes the need to look at the public methods of the component and to understand how to call them in order to properly integrate the component, since this information is obtained through the tasks model.

The purpose of this paper is to propose means that can help this integration of tasks models and the merging of parts of them. The solution we propose is inspired by results obtained in the domain of XML tree composing applied to the merging of graphical user interfaces. We now introduce these previous results before presenting how we use them for TO components merging.

## 3 XML Tree Composing

A tasks model can be expressed in XML. The XML document can be associated to algebra tree. We propose to use the tree algebra to manipulate the XML document in which the tasks model is written. The TAX model (Tree Algebra for XML) [6] defines a data tree as a rooted, ordered tree, such that each node carries data (its label) in the form of a set of attribute-value pairs. Each node has a special, single valued attribute called ***tag*** whose value indicates its type. A node may have a ***content*** attribute representing its atomic value. Each node has a virtual attribute called ***pedigree*** drawn from an ordered domain. The pedigree carries the history of “where it comes from”. Pedigrees play a central role in grouping, sorting and duplicate elimination.

Originally proposed for database management, this model is also well suited to manipulate XML documents from Human-Computer Interaction domain. In [7], for instance, some operators (such as Union, Fusion, Selection, Difference, Equals operators) have been adapted in order to manipulate Graphical User Interfaces (GUIs) defined with the UsiXML UIDL [10] based on XML. Besides, a plug-in (CompositXML) has been developed for the GrafiXML editor to compose GUIs [7].

The following example illustrates how this principle of XML tree composing based on the TAX model and applied to GUIs works. The Figure 2 shows the tree representation of a Union operator applied on two input interfaces. This result is operated from two XML trees in the case of horizontal union (this layout precision is specific to the Concrete User Interface Operator). The input interfaces and the resulting one (Final User Interface on java platform) are presented in the Figure 3. The Union operator creates a new window, whose width is equals to the sum of the two input windows’ width. It also adds a box in the tree to indicate the new type of layout (horizontal in this example). Then the duplicates are deleted; for instance, as the two buttons *Save* and *Close* appears on each form, they will be deleted from one of them (this choice — which ones will be deleted — is made by the user). To do this, the algorithm uses the tag “default value” and the content associated to compare the element. It uses the pedigree to know if the parent elements are repetitive. In the example, the box (with type = horizontal) is repetitive if the children are two buttons with default value equals to *Save* and *Close;* to do that, the pedigree is used.

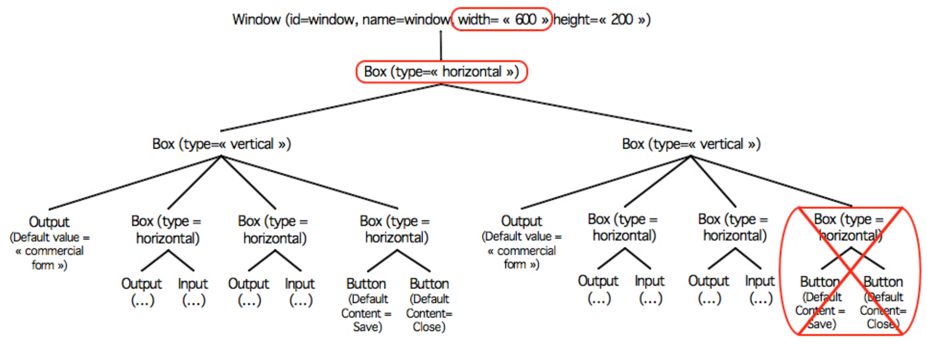

Window (id=window, name=window width= « 600 » height=« 200 »)
Box (type=« horizontal »)
Box (type=« vertical »)
Box (type=« vertical »)
Output (Default value = « commercial form »)
Box (type = horizontal)
Box (type = horizontal)
Box (type = horizontal)
Output (...)
Input (...)
Output (...)
Input (...)
Button (Default Content = Save)
Button (Default Content= Close)
Output (Default value = « commercial form »)
Box (type = horizontal)
Box (type = horizontal)
Box (type = horizontal)
Output (...)
Input (...)
Output (...)
Input (...)
Button (Default Content = Save)
Button (Default Content= Close)

**Fig. 2.** The tree resulting from the application of a Union operator on two interfaces.

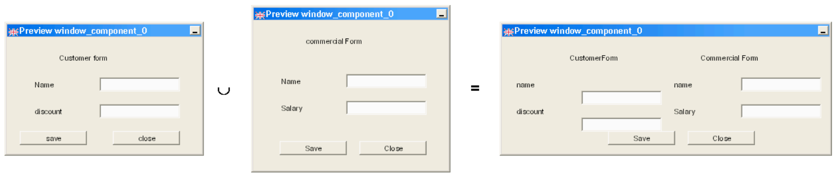


**Fig. 3.** Union of two user interfaces without repetition of common part. The resulting interface is described by the tree of the Figure 2.

We think that this principle based on tree algebra can also be applied at a higher level, at the tasks model level. That is what we now illustrate through an example, showing how the XML tree composing approach can assist the TO components composition problem.

## 4 Tasks Models Merging Using Composition Operators

### 4.1 The composition problem

First, we start from the assumption that we have at our disposal two components — a chat tool and a shared whiteboard — that have been developed according to the Task-Oriented design approach. The architecture of each of these two final stand-alone components is then similar to the one illustrated on the Figure 1. As we said in this previous part, the use of tasks models should ease the composition of such components. Indeed, assembling such components, in which the tasks model is linked to the functional code, can then be realized through the integration, or linkage, of their individual tasks models in a more global tasks model, the one of the global environment. As each individual tasks model is linked to the code of the component it describes, the global environment will be able to know which methods to call in order for this component to be properly integrated. The introduction of the composition principle based on tree algebra will provide assistance for realizing this integration, by helping to merge several subtasks.

The embedded tasks models of these two high-level components appear in the Figure 4. Both may include a specific and similar connection subtask corresponding to a specific interface asking the user for his/her login and password (and eventually other information specific to each component, like the channel for the chat, or the board to

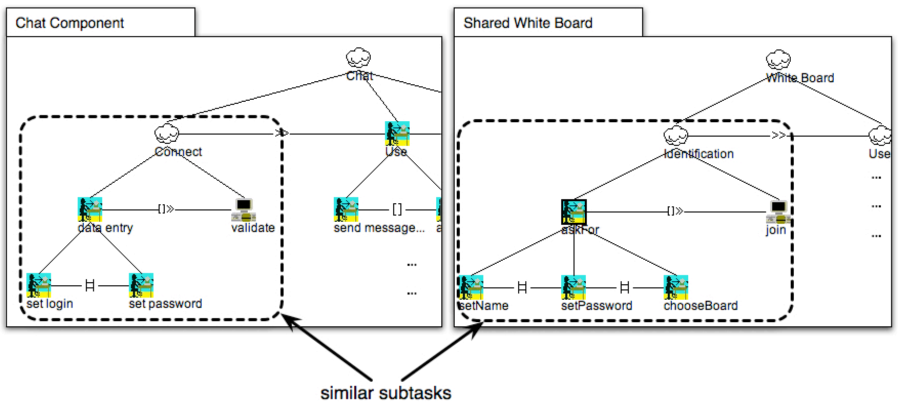


**Fig. 4.** Similar parts between the Chat tasks model and the White board tasks model. Thanks to the algebra tree composition approach, we want to assist the merging of these two subtasks so that the global environment integrating the two components may take in charge simultaneously the two connection processes.

join for the whiteboard). If we integrate these tools in the global environment without any specific merging, the environment will just launch the tools without any particular configuration, and the user will have to identify him/herself twice (once for the chat, and once for the white board, since they both have their own similar "connect" or "identification" subtask which is required). We propose a mechanism in order to assist the integration of these tasks models by merging the appropriate subtasks.

## 4.2 Tasks models composing using tree algebra

We can imagine several scenarios of composition. In this paper, we focus on the following one: we want the global environment to do the connection phase (that means the "connect" subtask for the chat, and the "identification" subtask for the whiteboard) at the very beginning; after that, the user will be able to use both tools in parallel. If we look at the tasks models of these two components (see Figure 4), we understand that assisting this scenario will consist in: 1) extracting the two subtasks or subtrees tied to the connection processes; 2) merging these two subtasks in one; and finally 3) plug into the global tasks model the three resulting sub-models (the one containing the result of the merging, and the tasks models of the two components). In order to realize this, we adapt the tree composing approach presented before.

The tasks models are described and stored in XML documents. The first step consists in transforming them in XML trees. The Figure 5 illustrates this transformation applied to the tasks model of the chat component. These XML trees serve as the basis to the assisted composition. According to our scenario, we want to merge the *Connect* subtask of the chat and the *Identification* subtask of the whiteboard. From the tree algebra and the work we have presented in part 3 about GUI composition, it corresponds to the Union operator.

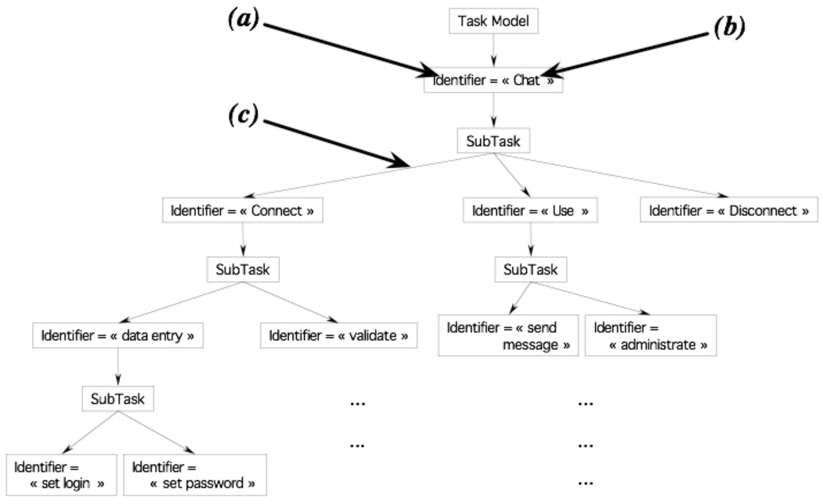

(a)
Task Model
(b)
Identifier = « Chat »
(c)
SubTask
Identifier = « Connect »
Identifier = « Use »
Identifier = « Disconnect »
SubTask
SubTask
Identifier = « data entry »
Identifier = « validate »
Identifier = « send message »
Identifier = « administrate »
SubTask
...
Identifier = « set login »
Identifier = « set password »

**Fig. 5.** Part of the XML tree obtained by transformation of the tasks model of the chat component. Some examples of the key concepts used during tree algebra transformation appear on it: ***(a)*** a *tag*, ***(b)*** a tag's *content*, and ***(c)*** a *pedigree*.

There are many ways to implement a Union operator between two trees. The more immediate possibility consists in creating a new tree where the root's identifier is equal to the name of the global task; then a subtask is created and the two input trees (corresponding to the tasks models of the two TO components) are just plugged into this subtask. However, this basic solution does not do any merge between the similar connection subtasks of each component; it does not manage the problem of the repetitive connection process. A second alternative consists in deleting one of the two similar subtasks. It was the choice made for the implementation of the Union operator in the frame of GUIs composition (cf. part 3). In this case, two solutions are possible: either we choose to delete the connect subtask of the Chat, or the identification subtask of the shared whiteboard. But in both solutions, a problem remains: in the global environment, the user should be able to use the two components in parallel. If we delete one of the two similar subtasks — the connect subtask of the Chat for example — and do every necessary connection in the remaining one — the identification subtask of the whiteboard —, the resulting tree is not coherent; indeed, the Chat task may be initiated before the connection which stands in the whiteboard. The *enabling* relationship between the "connect" and the "use" subtasks of the Chat forbids this solution. The other solution — keeping the Chat connection and removing the whiteboard identification — presents the same problem.

The algorithm we propose in order to implement the Union operator between two tasks trees is the following. It first creates a new sub tree that will contain the merging of the two similar subtasks. This merging is possible since both connection subtasks have a similar structure. The algorithm imports one of them in the new tree, and adds to it the missing subtasks appearing in the second one, according to their order in the corresponding input tree. In order to respect the enabling relationship between the connection subtask and the use subtask of each component, the created task must be placed before the integration of the components' individual tasks models. Once the two subtasks have been merged, only one interface will be presented to the final user at the beginning, and s/he will fill the form only once. The task generated by the merging takes in charge the necessary methods calls on the two components in order for their own "connect" subtasks to be effectively realized. This is possible because each TO component contains the links existing between some tasks of their model and specific methods of their code (cf. part 2). The final resulting functioning is illustrated on the Figure 6 by the tasks model of the global environment that integrates the result of the merging process and the tasks models of the two components. As we can notice, the enabling relationship is kept between the merged subtask that realizes the connection processes and the parallel use of the two TO components.

This simple example illustrates what kind of merging this approach is able to realize, depending on the choices of the user that will finally do the component assembling. This proposition assists the integration of individual tasks models. If the person in charge of this integration specifies which subtasks are similar and should be merged (in our example, the two "connect" subtasks), our solution is able to merge these subtasks in one task in the global model, and eventually merge the corresponding interfaces too.

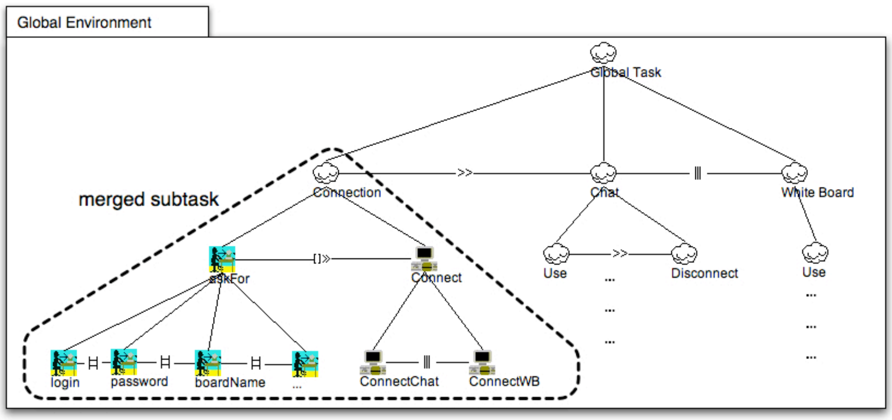


**Fig. 6.** Part of the tasks model of the global environment after the merging process. It contains first the tasks model resulting from the merging process, and the two adapted tasks models of the integrated components.

## 5 Conclusion and Future Work

Providing means that will allow users to adapt their software environments by finely and dynamically integrating high-level components is truly a challenge. A first problem stands in the fact that components are generally hardly understandable. We have presented the Task-Oriented approach that proposes a new way to construct more understandable high-level components, especially thanks and through the use of tasks models. Assembling TO components means then, according to this approach, composing tasks models. In order to assist this integration, we have proposed a mechanism inspired by previous work on tree algebra applied to GUIs composition. TO components' individual tasks models are described in XML documents. Assembling tasks models can then be seen as assembling XML trees. Tree operators can then be adapted to this specific domain. We have illustrated this approach with an example: the integration of two TO components with the automatic merging of two similar subtasks they share. The tasks models are transformed into trees, on which we apply a Union operator that merges the identified similar subtasks. The resulting tasks models are then integrated in the global model of the integrating environment. Even if this example seems to be very specific, it can be extended to other operators and technologies. The only requirement is that the high-level components have to be developed according to the TO approach. We now pursue our efforts in order to generalize this approach and provide a more efficient assistance to high-level component composition by users.

## Acknowledgments

The present research work has been supported by the “Ministère de l'Education Nationale, de la Recherche et de la Technologie”, the “Région Nord Pas-de-Calais” and the FEDER (Fonds Européen de Développement Régional) during the projects MIAOU and EUCUE. The authors gratefully acknowledge the support of these institutions. The authors thank also Jean Vanderdonckt for his contribution concerning UsiXML.

## References

1. Blevins, D.: Overview of the Enterprise JavaBeans Component Model. In [5] (2001) 589–606
2. Clerckx, T., Luyten, K., Coninx, K.: The Mapping Problem Back and Forth: Customizing Dynamic Models while preserving Consistency. TAMODIA 2004, 15-16 November, Prague, Czech Republic (2004)
3. Cubranic, D., Murphy, G.C., Singer, J., Booth, K.S.: Learning from project history: a case study for software development. In Proc. of CSCW04. ACM Press (2004) 82–91.
4. Ferris, C., Farrel, J.: What are web services? Comm. of the ACM, Vol. 46(6) (2003) 31
5. Heineman, G.T., Councill, W.T. (eds.): Component-based software engineering: putting the pieces together. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)
6. Jagadish, H.V., Lakshmanan, L.V.S., Srivastava, D., Thompson, K.: TAX: A Tree Algebra for XML. Lecture Notes In Computer Science, Vol. 2397 (2001) 149–164
7. Lepreux, S., Vanderdonkt, J., Michotte, B.: Visual Design of User Interfaces by (De)composition. In Proc. of DSV-IS 2006 (Dublin, Ireland, July 26-28, 2006). Springer, Lecture Notes in Computer Science (2006)
8. Lewandowski, A., Bourguin, G.: Inter-activities management for supporting cooperative software development. In Nilsson et al. (eds.): Advances in Information Systems Development: Bridging the Gap between Academia and Practice, Vol. 1. Springer Verlag (2005) 155–167
9. Lewandowski, A., Bourguin, G., Tarby, J.C.: Les Modèles de Tâches pour la Contextualisation des Composants. Proc. of the 10th Intern. Conf. Ergo'IA, Bidart/Biarritz, France, 11-13 October (2006) 147–154
10. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López, V.: UsiXML: a Language Supporting Multi-Path Development of User Interfaces, Proc. of EHCI-DSVIS'2004 (Hamburg, July 11-13, 2004), Lecture Notes in Computer Science, Vol. 3425. Springer-Verlag, Berlin (2005) 200–220.
11. Suchman, L.: Plans and Situated Actions. Cambridge University Press (1987)
12. Van der Aalst, W.: Don't go with the flow: Web services composition standards exposed. Trends Controversies Jan/Feb 2003 issue of IEEE Intelligent Systems (2003)
13. Wang, N., Schmidt, D.C., O’Ryan, C.: Overview of the CORBA component model. In [5] (2001) 557–571