Towards a UML profile for the description of dynamic software architectures

Mohamed Hadj Kacem[1], Mohamed Nadhmi Miladi[1], Mohamed Jmaiel[1], Ahmed Hadj Kacem[1], and Khalil Drira[2]

[1] University of Sfax, laboratory LARIS-FSEGS, B.P. 1088 Sfax, Tunisia,
mohamed.hadjkacem@fsegs.rnu.tn
[2] LAAS-CNRS, 7 Avenue du Colonel Roche 31077 Toulouse, France

Abstract. In this paper, we propose a unified approach based on visual notations for describing dynamic component-based software architectures. Our approach allows describing the static, the dynamic and the behavioral aspect as well as the architectural constraints to be respected during the architecture evolution. We specify, using UML2.0, the static aspect of a software architecture in accordance with an architectural style, the dynamic aspect using the graph transformation rules, and the behavioral aspect based also on the UML2.0 notation. These specifications are defined according to the proposed notation integrating UML2.0 and OCL language. Indeed, all constrains, all functional and some structural actions can be expressed using the OCL language. These three aspects offer to the architects an intuitive and complete way to specify the software architecture.

1 Introduction

The dynamicity criteria gain more and more importance in software architectures design. Such dynamic architectures represent systems that do not simply comply a fixed and static structure, but can react and evolve to certain requirements or events at run-time reconfiguration of its components and connections. Indeed, during its life cycle, the dynamic architecture evolves in various manners: by activating and/or deactivating components and by building and/or deleting connections [1].

The architecture evolving must be achieved without disturbing the services provided by the application. The adaptability of these architectures to these changes is a recent requirement which needs a frequent evolution and a *dynamic configuration*.

This has constituted for a few years an active research field and development articulated around the study of *dynamic software architectures* [2], [3], [4], [5], [6]. The principal objective is thus to model and to control the evolution and the reconfiguration of architectures by expressing their conformity compared to a given *architectural style*.

Our research enters in this framework. In this paper, we suggest a denotational approach based on a unified chart. It allows to describe the static, the

dynamic and the behavioral aspect as well as the architectural constraints to be respected during the system evolution.

Our approach seeks to take advantage of the expressive capacity of several notations and formalisms such as UML2.0 [7], OCL (Object Constraint Language)[8], [9] and graph grammars [10]. Thus, we describe the static aspect of software architecture, in accordance with an architectural style, with UML2.0 notation. We express the dynamic aspect with a new notation, based on UML2.0, representry graph rewriting rules. We describe the behavioral aspect with a notation based again on UML2.0. To describe the functional part: the application pre-condition which aims to control the evolution in accordance with the system properties, we use the OCL languages.

This integration, based on UML2.0 notation, offers to the architects an intuitive specification technique, easy to apprehend and understand and which presents a high power of expressiveness. The transformation rules take into account the structural and functional constraints ensuring thus the system consistency during its evolution.

This paper is organized as follows: In section 2, we present a discussion of related work. We briefly examine the advantages and the disadvantages of the ADLs as well as the formal approachs and the UML2.0 notation. We highlight their capacities to describe the static and the dynamic aspects of software architectures. Section 3 details our approach which describes the software architecture by considering the static, dynamic and behavioral aspect. This approach is illustrated with an example. Ultimately, in section 4, we present a conclusion and the perspectives of our work.

2 Related work

The study of the description techniques and the reconfiguration of software architectures enables us to identify mainly three types of research.

2.1 ADLs

ADLs or (Architecture Description Languages) [11], [12] emerge as a solution and respond to these problems partly by allowing the definition of a precise and common vocabulary for the actors having to work around the software architecture specification. They offer a well-defined set of notations for the description of an application architecture. Basically, an ADL allows the developer to describe the abstract organization of his system in terms of coarse-grained architectural elements such component, connector, port and configuration or architecture [13].

ADLs formalize software architectures by offering a simple, yet flexible, representation. An ADL may comprise a formal or semi-formal descriptive textual language, a graphical language, or both. ADL provides the conceptual framework for representing software at the architecture level. Most ADLs were developed to focus on specific aspects of software architecture.

We noted that the majority of ADLs are concentrated on the static description of architectures but the dynamic of architecture is not enough expressed. Our study [11] enabled us to identify several weak points in ADLs. These weak points are in several orders. Indeed, the dynamic of architectures is not well supported by ADLs although there are some proposals which are interesting Darwin [14] and Olan [15]. Moreover, the dynamic of software architecture is not much expressed. ADLs are interested only in the systems having a fixed number of configurations which have to be known in advance Wright [16]. Thus, it is not possible to perform arbitrary reconfiguration operations and particularly those which are external during the application execution.

In addition, except the Wright language, ADLs based approaches do not allow to distinguish various architectural styles. Finally, most of them are not formally defined. This prevents rigorous analysis and verification of architectural properties.

Finally, we can say that the dynamic aspect is not well supported by ADLs. In spite of the great number of interesting proposals, these solutions are not enough thorough. The majority of ADLs, are able to manage only systems having a finite number of configurations which should be known in advance. Then, it is impossible to arbitrarily execute reconfiguration operations during run-time.

2.2 Formal approaches

The formal approaches to specifying dynamic software architectures that we consider fall into four categories: graph-based approaches [10], [17], [18], process algebra approaches [19], [20], logic-based approaches[21], [22], and other approaches [23], [24]. In this paper, we are interested in graph-based approaches and particularly in the graph transformation.

To specify the software architectures and the architectural styles it is possible to use a graph grammar to represent the style and a graph to represent a specific system's architecture. Graphs are not only a common formalism in the specification of static software architectures but can also describe dynamic architectures. They represent the most intuitive mathematical formalism for modeling structures. In this context, [10] proposes to represent the software architecture using a graph and the architectural style with a context-free graph grammar. This kind of grammars does not allow to describe certain logical properties which permit, for example, to reason about the instance number of a given component. Reconfiguration is described with a set of rewriting rules whose definition is rather simple and comprehensible. Theses rules explain clearly the topological changes, but they do not allow to express certain logical conditions such as the absence of a communication link between two software components. [23] proposes an approach for formal specification of dynamic architectures of component based systems. This approach is based on an integration of graph-based semantics in the framework of the Z formal language. This facilitates the task of the developer by offering a specification technique which is easy to apprehend and which enables him to rigorously reason about architectural styles. Furthermore, this

approach allows to formally describe the dynamic of a software architecture using graph rewriting rules. These rules take into account the properties specifying the constraints for performing reconfiguration operations which ensures the consistency of a system during its evolution. This approach describes the static, the dynamic aspect and the functional constraints but it requires a knowledge and an expertise of Z notation and graphs grammars.

Graph transformation is applied to describe the evolution of software architecture in a formal way. But like the majority of the formal approaches, graph transformation is based on mathematical notations which requires a knowledge and an expertise by users.

2.3 UML

Several works seeks to describe the evolution of software architectures by using UML. Research to represent architecture in UML can be approached in two ways. The first called “as is”. It consists to using the existing UML notation to represent the software architectures. The second extends the vocabulary and semantics of UML to match its modeling capabilities to the concepts of architectures. The second way can be divided again into “heavyweight” and “lightweight” way. The heavyweight way [25] adds new modeling elements or replaces existing semantics by directly modifying the UML metamodel. The lightweight way [26], [27] defines new modeling elements by means of the extension mechanism of UML but does not modify the UML metamodel.

To conclude, and according to the studies that we achieved, we noted that UML allows to describe the static aspect of a software architecture in accordance with an architectural style. UML, and in spite of the various offered diagrams, does not allow describing the dynamic of software architectures. Indeed, based on the offered notation, it is impossible to follow the architecture evolution in terms of creation and removal of components and/or connections. This is why, several research works [2], [17], [28] seek to combine UML with other formalisms in order to support the dynamic of the software architecture description.

2.4 Synthesis

According this study, we notice that ADLs, the graph transformation and the UML language contribute, each one of its manner, to the specification of software architecture reconfiguration. The table 1 gives a short outline on their principal advantages and limits.

According this study, we notice that graph transformation and UML language are complementary. Indeed, the limits of the one are the advantages of the other. In the following, we present an approach which integrates the two notations and thus, allows to benefit from the advantages of the two approaches and to cover their limits.

Table 1. Synthesis.

	ADL	Graph transformation	UML
Advantage	- Languages proposing coherent solutions to solve the reconfiguration of component-based architectures.	- A powerful notation for modeling dynamic architectures. - Represent an intuitive mathematical tool for modeling structures.	- Language very used. - Standard. - Provide good means to model component-based software architectures (contribution UML2.0).
Limit	- Multi notations and multi languages. - Absence of standards.	- Absence of standards. - Requires expertise.	- Weakness for modeling dynamic architectures.

3 The proposed approach

We tried in [29] to extend ADLs to describe the static aspect in accordance with an architectural style but by adopting an easier notation and known by the community. We have adopted the UML notation. To solve UML insufficiencies, we combined UML with graph grammars in order to describe the dynamic aspect. With this combination, we describe the dynamic of the system but the combined notation used is difficult to understand and requires an expertise in graph grammars.

We propose in this paper, a UML Profile allowing the description of the static, dynamic and behavioral aspect. In the static aspect, we describe the software architecture in terms of components and connections in accordance with an architectural style. In the dynamic aspect, we describe the architecture evolution in terms of creation and removal of components and/or connectors. In the behavioral aspect, we describe the coordination between the various reconfiguration operations.

To give more details on our approach, we develop in parallel, an illustrative example. The *Patient Monitoring System* (PMS), which was used to illustrate works of [23] and [10]. For each service of the private clinic (pediatric, cardiology, maternity, etc.) we associate an event service to manage the communications between nurses and bed monitors. For each bed monitor, the responsible nurse periodically requests patient data (for example, blood pressure, pulse and temperature) by sending a request to the event service to which it is connected. This service transmits the request to the concerned bed monitors. When a pa-

tient state is considered to be abnormal, its corresponding bed monitor raises an alarm to the event service to which it is connected. Then, this service transmits the signal to the responsible nurse. To represent the communication architecture of this system, we chose the *Producer/Consumer* style. So, the nurse and the bed monitor behave respectively as a consumer component and a producer component.

In addition to the architectural style constraints, an application can have specific properties which must be satisfied during the evolution of its architecture. We will take some properties of the PMS system such as:

1. The system must contain at most 3 services.
2. A service contains at most 5 nurses and 15 patients.
3. A patient must be always affected with only one service. This later must contain at least one nurse to take care of this patient.
4. A nurse must be connected to only one service.
5. A nurse cannot control more than 3 patients.
6. A patient can be controlled only by one nurse.
7. The existence of a patient implies the existence of a nurse.

3.1 Static aspect

The static aspect defines the type of components able to occur in the system description and the type of connections which can link these components. It defines also the set of the architectural properties which must be satisfied by all configurations belonging to this style. To specify the static aspect, we use the UML notation. Nodes represent components and arcs represent connection between these components.

The representation of the static aspect is made up of three parts as depicted in figure 1.

- *Style Name*: In this part, we describe the name of the system to specify.
- *Architectural Style*: In this part, we describe our system in accordance with an architectural style. We specify the set of components and connectors which constitute the system. The specification is described using UML2.0 notation.
- *Gards*: In this part, we describe the architectural constraints that system must respect throughout its evolution. These constraints are expressed according the OCL language (Object Constraint Language).

We return to our example and we describe the static aspect by considering the architectural style constraints and the stated specific properties. The figure 1 describes the PMS system architecture according to the *Producer/Consumer* style.

According to UML2.0, our system is described based on three components (service, patient and nurse). The system can contain at most 3 event services. Each service can contain between 0 and 5 nurses and can include between 0

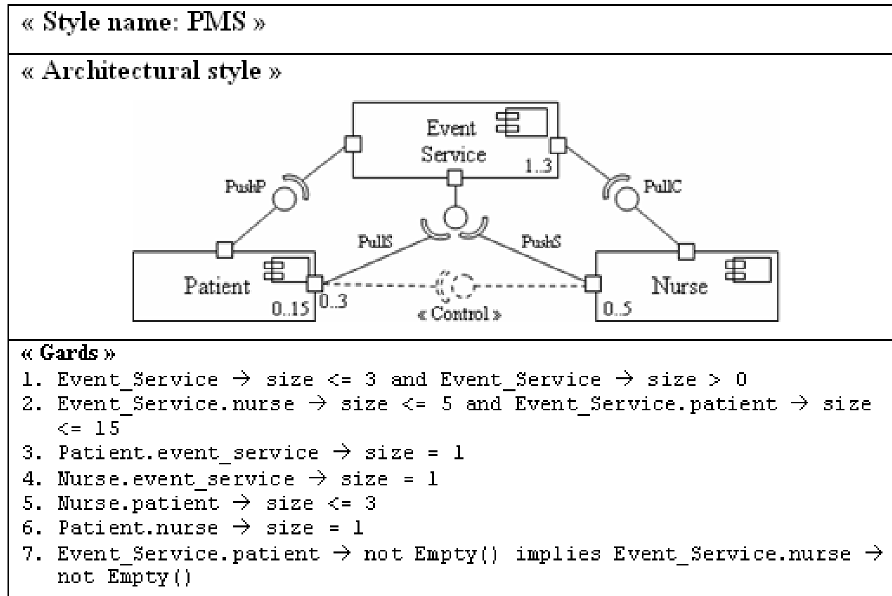

Fig. 1. Static Aspect of the PMS system

and 15 patients. A nurse cannot control more than 3 patients. To describe the connection between the patient and the nurse (constraint (5) and (6)) we extend UML2.0 with the stereotype ≪control≫ which represents just a semantic connection between two components. The figure 2 describes a possible configuration of PMS system.

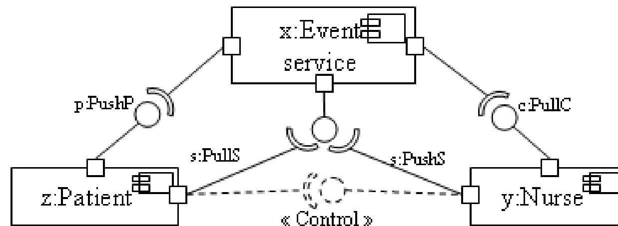

Fig. 2. One possible configuration of the PMS system

3.2 Dynamic aspect

The four fundamental reconfiguration operations, provided to describe the architecture evolution, are creation and removal (which can be also viewed as

activating and deactivating) of components and creation and removal of connections [1]. To describe these operations, we have defined a new notation described by the figure 3. This notation is based on graph transformation with the use of UML2.0 notation and the associated OCL language. This notation expresses the dynamic aspect through four sections:

- *Reconfiguration Operation*: In this part, we describe the name of the operation to execute.
- *Require & delete*: This part describes the system part removed during the reconfiguration operation.
- *Insert*: This part describes the fragment that is created and embedded into the system during the reconfiguration operation.
- *Require & preserve*: This part describes the system part that is identified but not changed during the reconfiguration operation.
- *Gards*: This part describes the architectural constraints the system must respect throughout its evolution. These constraints are expressed according the OCL language.

In the parts: "Require & delete", "Insert" and "Require & preserve", the specification of components and connectors which constitute the system is described by UML2.0 notation.

We return to our example and we present the specification of some rules allowing evolving our PMS system while taking into account the constraints of style architecture and the architectural properties.

Insertion of an Event_Service This rule allows to insert a component instance of type Event_Service. To apply this rule, we should check that the system does not already contain three Event Services according to the constraint OCL `Self→size<3`. The representation of this rule is depicted in figure 3.

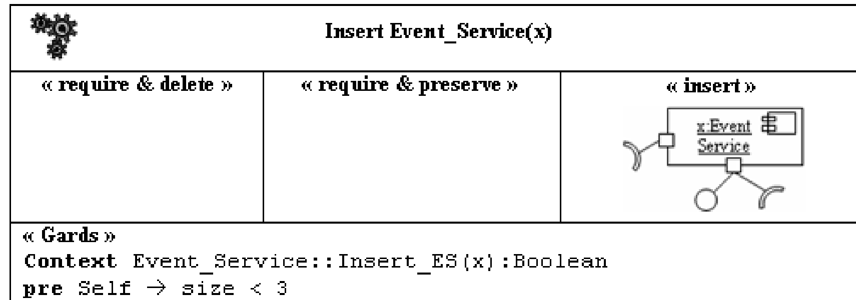

Fig. 3. The insertion rule of an Event_Service

The application of this rule to the architecture instance presented by figure 2 with `x':Event_Service` as parameter generates the graph depicted in figure 4.

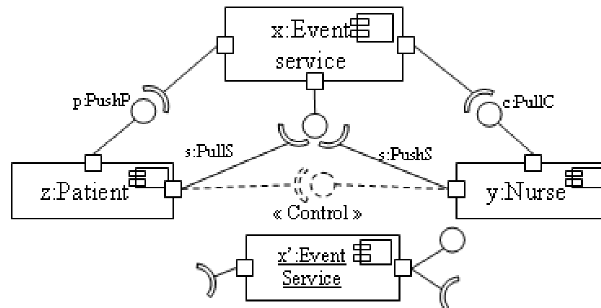

Fig. 4. Graph obtained following the application of Insert Event_Service to the graph of figure 2

Insertion of a nurse This rule allows to insert and to connect a Nurse to an Event Service. To apply this rule, we should check that the Event Service does not contain already five nurses in according to the constraint (2). The representation of this rule is depicted in figure 5.

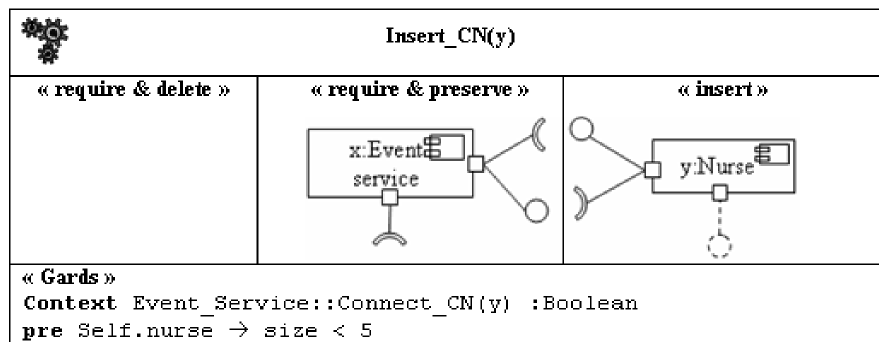

Fig. 5. The insertion rule of a nurse

The application of the rule Insert_CN to the architecture instance presented by figure 4 with `y':Nurse` as parameter generates the graph of figure 6.

Insertion of a patient This rule allows to insert and to connect a component instance of type Patient to an Event Service. To apply this role, we should check two conditions. First, it would be necessary that there is at least one nurse belonging to this service which may be charged with this new patient and the number of patients in load is lower than three. Second, we should check that this service does not already contain fifteen patients. The representation of this rule is depicted in figure 7.

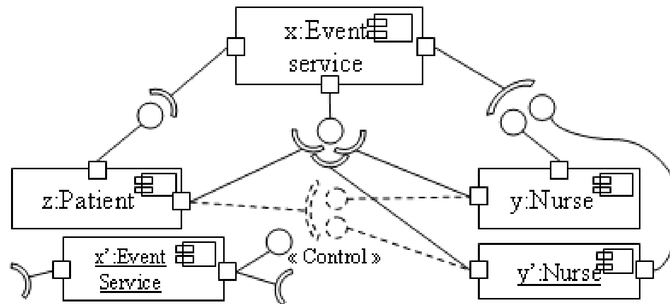

Fig. 6. Graph obtained following the application of Insert_CN to the graph of figure 4

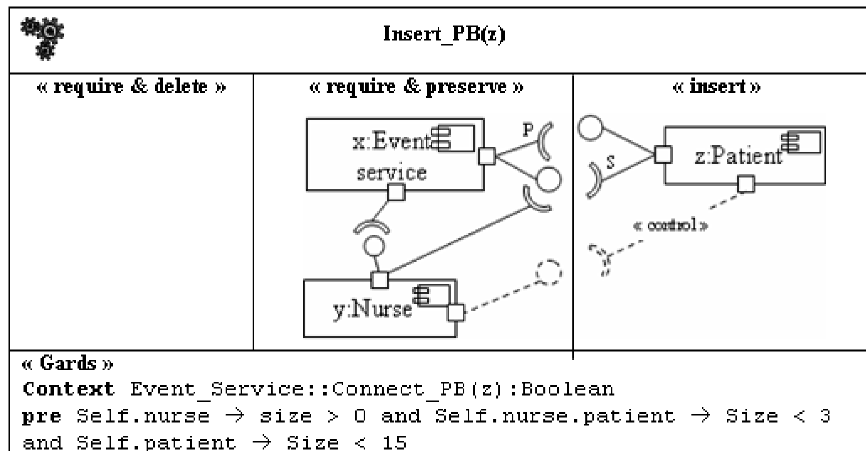

Fig. 7. The insertion rule of a patient

Transfer of a nurse A nurse can leave her service towards another (without being removed from the system) only if she is not responsible for any patient and the new service contains less than five nurses. The representation of this rule is depicted in figure 8.

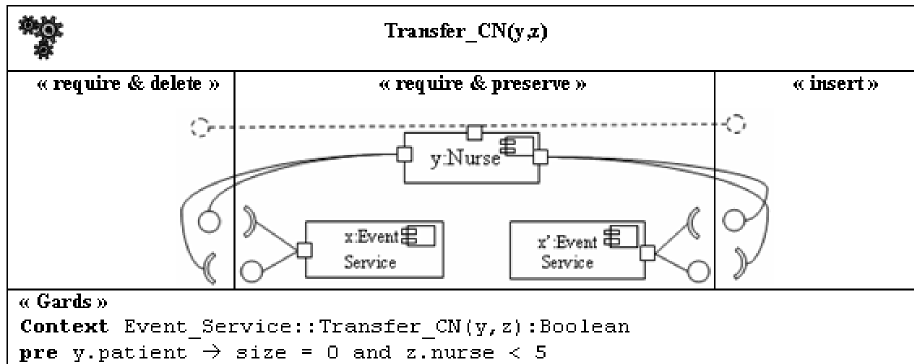

Fig. 8. The transfer rule of a Nurse

Transfer of a patient A patient can leave a service towards another (without being removed from the system) only if it is not treated with a nurse of the old service `y:Nurse` and if there is at least one nurse belonging to a new service `y':Nurse` which may be charged with this new patient `z.patient→size<3`. The representation of this rule is depicted in figure 9.

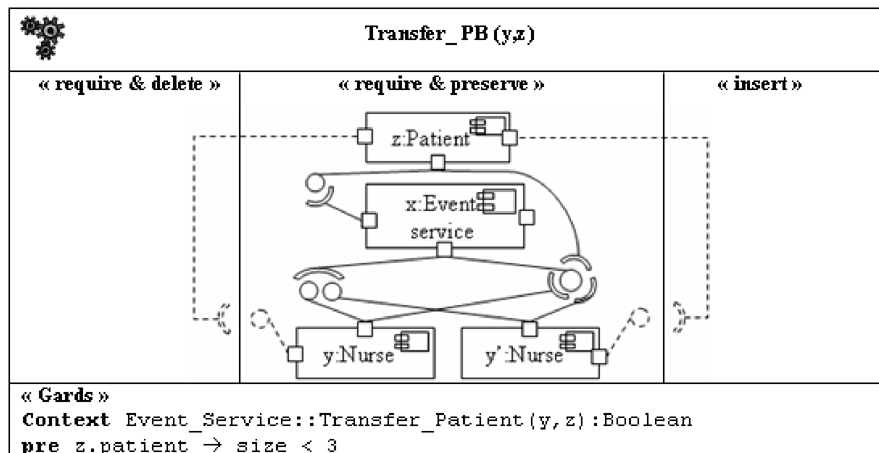

Fig. 9. The transfer rule of a Patient

3.3 Behavior Aspect

The behavioral aspect allows describing the sequence and the organization of the reconfiguration operations. It describes the execution time order. To describe the behavioral aspect, we use the activity diagrams.

According to the PMS specification depicted in figure 10, the first operation to execute is the insertion of the Event Service *"Insert Event Service"*. A nurse cannot be inserted *"Insert Nurse"* unless there is already an Event Service. A patient cannot be inserted *"Insert Patient"* in an Event Service unless there is already a Nurse connected to the service. The insertion of an Event Service, Nurse and Patient can be executed one or more time. The transfer of a patient *"Transfer Patient"* requires that this one be inserted and requires the existence of a nurse inserted to the Event Service in question.

4 Conclusion

We presented in this paper a unified approach based on graphic representation for describing the dynamic component-based architectures. This approach allowed, using the UML2.0 notation, describing the static aspect in accordance with an architectural style. It helps to specify the dynamic aspect of software architecture by integrating UML2.0 and OCL language. It allows also to describe the behavior and the relations between the configuration operations.

Our approach profits of the advantages of graph transformation and UML notations. The combination that we proposed covers their limits. This integration offers to the architects an intuitive specification technique, which can be easy to apprehend trough presenting a high expressive power. Our approach allows describing the dynamic of architectures via graph rewriting rules. The rules take into account the properties which specify the constraints that condition applying reconfiguration operations. This ensures the consistency of the system during its evolution.

We have planned, in prospect, to model the metamodel of the three aspects which we proposed and to implement the so-defined profile on the one hand and to establish the link between our work and that of [23] on the other hand. For this reason, we develop a translator from our visual specification notation into the Z formal notation.

PMS Behavior

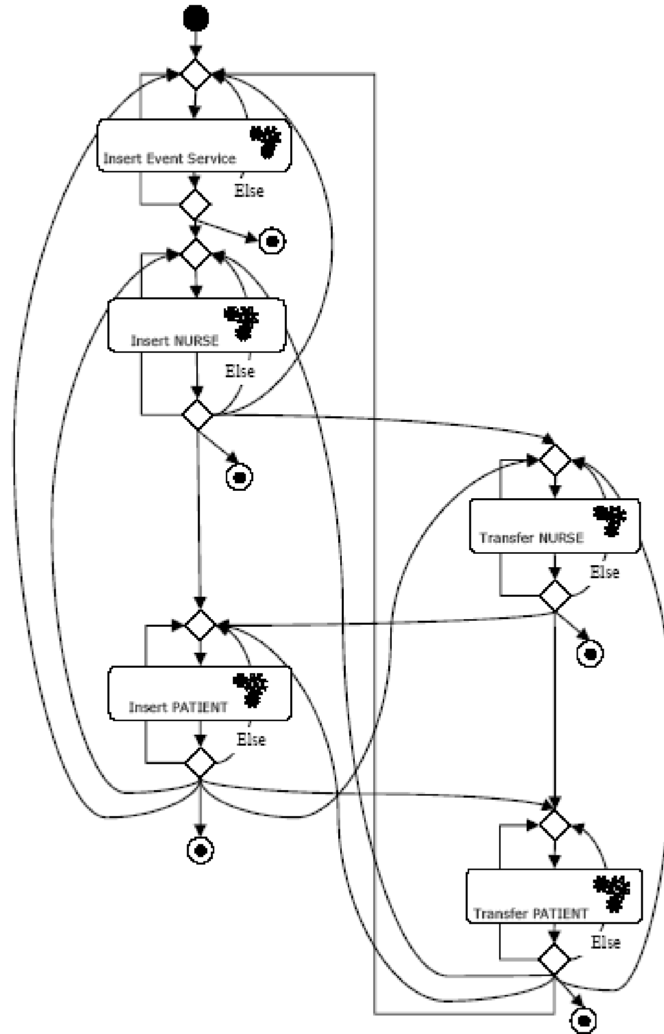

Fig. 10. The behavior of configuration operations

References

1. Wermelinger, M.: Specification of software architecture reconfiguration. Phd thesis, Université Nova de Lisbon (1999)
2. Baresi, L., Heckel, R., Thone, S., Varro, D.: Style-based refinement of dynamic software architectures. In: WICSA04: The 4th Working International IEEE/IFIP Conference on Software Architecture, Oslo, Norway, IEEE Computer Society (2004) 155–164
3. Rasche, A., Polze, A.: Configuration and dynamic reconfiguration of component-based applications with microsoft .NET. In: ISORC'03: The 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington, DC, USA, IEEE Computer Society (2003) 164
4. Oriol, M., Serugendo, G.D.M.: Disconnected service architecture for unanticipated run-time evolution of code. IEE Proceedings - Software **151** (2004) 95–108
5. Chaudet, C., Greenwood, R.M., Oquendo, F., Warboys, B.: Architecture-driven software engineering: Specifying, generating, and evolving component- based software systems. IEE Proceedings - Software **147** (2000) 203–214
6. Roh, S., Kim, K., Jeon, T.: Architecture modeling language based on UML2.0. In: APSEC, The 11th Asia-Pacific Software Engineering Conference, IEEE Computer Society (2004) 663–669
7. OMG: UML 2.0 superstructure specification, final adopted specification. Omg document (2003)
8. Akehurst, D., Patrascoiu, O.: OCL 2.0 - implementing the standard for multiple metamodels. Uml 2003 preliminary version, technical report, Computing Laboratory, University of Kent, Canterbury, UK (2003)
9. OMG: The unified modelling language 2.0 - object constraint language 2.0 proposal. OMG document, url: http://www.omg.org (2003)
10. Métayer, D.L.: Describing software architecture styles using graph grammars. IEEE Transactions on Software Engineering **24** (1998) 521–533
11. Hadj Kacem, M., Jmaiel, M., Hadj Kacem, A., Drira, K.: Evaluation and comparison of ADL based approaches for the description of dynamic of software architectures. In: ICEIS'05: The 7th International Conference on Enterprise Information Systems, Miami, USA, INSTICC Press (2005)
12. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26** (2000) 70–93
13. Ermel, C., Bardohl, R., Padberg, J.: Visual design of software architecture and evolution based on graph transformation. Electronic Notes in Theoretical Computer Science **44** (2001)
14. Magee, J., Dulay, N., Kramer, J.: Structuring parallel and distributed programs. IEEE Software Engineering Journal **8** (1993) 73–82
15. Bellissard, L., Atallah, S.B., Kerbrat, A., Riveill, M.: Component-based programming and application management with Olan. In: OBPDC'95: Object-Based Parallel and Distributed Computation. (1995) 290–309
16. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. (In: Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)) 21–37
17. Heckel, R., Cherchago, A., Lohmann, M.: A formal approach to service specification and matching based on graph transformation. Electronic Notes in Theoretical Computer Science **105** (2004) 37–49

18. Ziemann, P., Hölscher, K., Gogolla, M.: From UML models to graph transformation systems. Electronic Notes in Theoretical Computer Science **127** (2005) 17–33
19. Hilderink, G.H.: Graphical modelling language for specifying concurrency based on CSP. IEE Proceedings - Software **150** (2003) 108–120
20. Salaün, G., Allemand, M., Attiogbé;, C.: A method to combine any process algebra with an algebraic specification language: the p-calculus example. In: COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment, Washington, DC, USA, IEEE Computer Society (2002) 385–392
21. de Paula, V.C.C., Ribeiro-Justo, G.R., Cunha, P.R.F.: Specifying and verifying reconfigurable software architectures. In: PDSE. (2000) 21–31
22. Aguirre, N., Maibaum, T.: Formal design and development of a corba-based application for cooperative HTML group editing support. Hierarchical Temporal Specifications of Dynamically Reconfigurable Component Based Systems **108** (2004) 69–81
23. Loulou, I., Hadj Kacem, A., Jmaiel, M., Drira, K.: Towards a unified graph-based framework for dynamic component-based architectures description in Z. In: ICPS'04: The IEEE/ACS International Conference on Pervasive Services, IEEE Computer Society (2004) 227–234
24. Kandé, M.M., Strohmeier, A.: Towards a UML profile for software architecture descriptions. In: UML 2000 - The Unified Modeling Language. Advancing the Standard. 3th International Conference, York, UK. Volume 1939 of LNCS., Springer (2000) 513–527
25. Pérez-Martinez, J.E.: Heavyweight extensions to the uml metamodel to describe the c3 architectural style. SIGSOFT Software Engineering Notes **28** (2003) 5–5
26. Selonen, P., Xu, J.: Validating UML models against architectural profiles. SIGSOFT Software Engineering Notes **28** (2003) 58–67
27. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E.: Modeling software architectures in the unified modeling language. ACM Transactions on Software Engineering and Methodology **11** (2002) 2–57
28. Bordbar, B., Giacomini, L., Holding, D.: Uml and petri nets for design and analysis of distibuted systems. In: The 2000 IEEE International Conference on Control Applications, Alaska, USA (2000) 610–615
29. Hadj Kacem, M., Jmaiel, M., Hadj Kacem, A., Drira, K.: Using UML2.0 and GG for describing the dynamic of software architectures. In: ICITA'05, The 3th International Conference on Information Technology for Application, Sydney, Australia, IEEE Computer Society (2005)