Towards a unified graph-based framework for dynamic component-based architectures description in Z.

Imen Loulou, Ahmed Hadj Kacem, Mohamed Jmaiel
University of Sfax
Laboratory LARIS
B.P. 1088, 3018 Sfax, Tunisia
E-mails : Imen.loulou@tunet.tn, {Ahmed@fsegs, Mohamed.Jmaiel@enis}.rnu.tn

Khalil Drira
LAAS-CNRS
7 avenue de Colonel Roche
31007 Toulouse Cedex 4, France
E-mail : Khalil@laas.fr

Abstract

This paper proposes a model oriented formal approach for the specification and the verification of dynamic component-based architectures. This approach associates the expressive power of functional and structural approaches. On the one hand, we make use of the specification language Z to formulate the constraints made on the architectural style. These constraints have to be maintained during the system evolution. On the other hand, we describe the dynamic of architecture in terms of graph-rewriting rules. The obtained rules take into account structural and functional constraints of the system under their application conditions ensuring in this way its consistency during its evolution. We express the rules entirely with the Z notation also obtaining, in this way, a unified approach which treats the static as well as the dynamic aspect. To validate our specifications, we use Z-EVES which is an advanced analysis tool supporting the Z-specification language.

Keywords: *software architecture, architectural style, dynamic architecture, graph rewriting, component-based applications, formal specification.*

1 Introduction

Currently, the computer science industry and research, in general, and software in particular, have to deal with software applications whose complexity continuously increases. This evolution forces the designers of the new applications to take into consideration the reinforcement of traditional applicative requirements. Moreover, they have to introduce new requirements of applications allowing creation and dynamic distribution of components and their interaction channels. The adaptability of the applications with respect to these changes is a recent prerequisite which requires a dynamic configuration and a frequent evolution of software architectures. This constitutes, since few years, a research field for the study of dynamic software architectures. The main objective is to control the evolution and the reconfiguration of architectures by maintaining their conformity with respect to a set of structural properties or architectural style.

Our study of techniques for software architecture description and reconfiguration allowed us to identify two general classes of research. The first one deals with Architecture Description Languages (ADLs). The most prominent among these languages are Darwin [1], Rapid [2, 3], Wright [4] and LEDA [5]. They provide modelling tools which help the designers to structure a system and to compose its elements. Generally, ADLs allows to describe only predefined dynamicity. That is, they are interested only in systems having a finite number of configurations which must be known in advance. In addition, except the Wright language, ADLs-based approaches don't allow to distinguish various architectural styles. Finally, most of them are not formally defined. This prevents rigourous analysis and verification of architectural properties.

The second class makes use of formal methods for specifying and verifying software architectures. We noted that

0-7803-8577-2/04/$20.00 ©2004 IEEE

these researches are almost focused on two kinds of formalisms : functional languages and graph grammars. Functional languages introduce abstract notations allowing to describe dynamic software architectures in terms of properties. For instance, temporal logic was used in [6] for its high expressive power and formal reasoning. Nevertheless, this logic is an axiomatic formalism which is difficult to use and requires, therefore, qualified experts. Moreover, reconfiguration is not well exhibited with this formalism. In [7], the authors suggest to specify architectural styles with the Z notation. However, this work concerns systems with static architecture. Graph grammars consist in using graphs for representing software architectures. They represent the most intuitive mathematical tool for modelling complex situations. In this context, [8] proposes to represent the software architecture using a graph and the architectural style with a context-free graph grammar. This kind of grammars doesn't allow to describe certain logical properties which permit, for example, to reason about the instances 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 don't allow to express certain logical conditions such as the absence of a communication link between two software components.

In this paper, we propose an approach which benefits from the expressive power advantages of the functional languages. It also exploits graph grammars privileges mainly in manipulating architectures. Hence, we describe a software architectural style using the Z notation [9] and reconfiguration operations using graph rewriting rules specified in the same notation (Z), thus obtaining a unified approach. These rules take into account both structural and functional constraints of a system in their application conditions ensuring, in this way, its consistency during its evolution.

Our approach has the advantage of providing a unified framework to formally specify both static and dynamic aspects of a software architecture. This ensures a coherence between the two aspects. In addition, Z notation is rather easy to use, while being sufficiently expressive, which facilitates the task of the developer. We can then use a theorem prover which supports Z specifications such as $Z/EVES$ [10] and *Z-HOL* [11].

This paper is organized as follows: in section 2, we describe the proposed approach and its main elements. Section 3 presents a case study which illustrates our approach. It also explains the verification proofs to be performed. In section 4, we give the concluding remarks and future work perspectives.

2 The proposed approach

Taking into account the relevance of graphs in the representation of structures [12] and their mathematical background, we use them to represent the software system architecture in a similar way to the approach presented in [8], where the nodes represent the components of the system and the arcs describe the communication links between these components. An architecture graph represents a given configuration of a system. This later may evolve by modifying its architecture.

In fact, the configurations or architectures represent instances of the architectural style of a system. This one describes the kinds of components and the kinds of relations which can connect them as well as the architectural properties which must be satisfied by all configurations belonging to this style [4].

We present in the following how to specify an architectural style in Z according to the general definition of an architecture graph. We will focus on directed labelled and typed graphs. In general, an architecture graph is defined as: $G=\langle N, L, E, T, f\rangle$ where:

1. $E \subseteq N \times L \times N$, is a set of edges ;
2. N is a set of nodes ;
3. L is a set of labels ;
4. $T=\{t1, t2, ..., tn\}$ is a set of types ;
5. $f{:}N \longrightarrow T$ is a mapping function which associates to each node one and only one type.

According to this definition, the associated Z Metamodel is represented by a schema, which consists of two parts: a declaration of variables and a predicate constraining their values.

It should be stressed that at this level of abstraction, we have no need to consider the representation of nodes of type t_i, so we introduce them as given sets: $[N\_t1, N\_t2, ..., N\_tn]$. So that, N_t1 ,for example, denotes the set of nodes of type $t1$.

system_name

$N_1 : \mathbb{F}\, N\_t1;\ N_2 : \mathbb{F}\, N\_t2;\ ...;\ N_n : \mathbb{F}\, N\_tn$
$R_1 : N\_ti \leftrightarrow l_j \leftrightarrow N\_tk$
$R_2 : N\_ts \leftrightarrow l_k \leftrightarrow N\_tj$
...
$R_m : N\_tu \leftrightarrow l_v \leftrightarrow N\_ts$

With:

- $N = \bigcup_{i=1}^{n} N_i$
- $E = \bigcup_{i=1}^{m} R_i$

- T=$\{t1, t2, ..., tn\}$
- L=$\{l_1, l_2, ..., l_n\}$

Let us take for example the Producer/Consumer style. The definition of an architecture graph belonging to this style is as follows:

$$G = \langle N, L, E, T, f \rangle$$

where:

- L=$\{pushP, pushS, pullC, pullS\}$;
- T=$\{Producer, Service, Consumer\}$;
- $E = \bigcup_{i=1}^{4} E_i$;
- $E_1 \subseteq \{x \in N \mid f(x) = Producer\} \times \{pushP\} \times \{y \in N \mid f(y) = Service\}$;
- $E_2 \subseteq \{x \in N \mid f(x) = Service\} \times \{pushS\} \times \{y \in N \mid f(y) = Consumer\}$;
- $E_3 \subseteq \{x \in N \mid f(x) = Consumer\} \times \{pullC\} \times \{y \in N \mid f(y) = Service\}$;
- $E_4 \subseteq \{x \in N \mid f(x) = Service\} \times \{pullS\} \times \{y \in N \mid f(y) = producer\}$;

According to this definition, the specification of this style in Z is given by the following schema:

$Prod\_Cons[N\_Producer, N\_Service, N\_Consumer]$

$P : \mathbb{F}\, N\_Producer$
$S : \mathbb{F}\, N\_Service$
$C : \mathbb{F}\, N\_Consumer$
$pushP : N\_Producer \leftrightarrow N\_Service$
$pushS : N\_Service \leftrightarrow N\_Consumer$
$pullC : N\_Consumer \leftrightarrow N\_Service$
$pullS : N\_Service \leftrightarrow N\_Producer$

This specification constitutes a generic schema for a software architecture. So, $N\_Producer$, $N\_Service$ and $N\_Consumer$ may be instantiated with any component sets. This schema may be used whenever we wish to introduce two objects that are related in this way.

The four fundamental reconfiguration operations, provided by almost all languages and systems, are creation and removal of components and connections [13]. Graph grammars showed their relevance to express these operations of reconfiguration in particular in the works described in [13] and [8]. The principle consists in specifying the subgraph witch will be replaced, how it is related to the rest of the graph, by which graph it will be replaced and how this new graph will be connected to the remaining part of the host graph. Generally, a rewriting rule is written: $g_l \longrightarrow g_r$ (g_l : graph left and g_r : graph right).

The study that we carried out on the graph rewriting techniques enabled us to appreciate the transformation principle of the Δ notation proposed in [14]. This notation expresses the structural constraints (existence/absence of a motif) via four sections:

- *The retraction* : the fragment of the graph removed during the rewrite.
- *The insertion* : the fragment that is created and embedded into the graph during the rewrite.
- *The context* : the fragment that is identified but not changed during the rewrite.
- *The restriction* : the fragment of the graph that must not exist for the rewrite to occur. If the subgraph matching the context and retraction can be extended to match the restriction, then the production cannot be applied to that subgraph

A Δ-rule r is applied to a graph g by the following steps:

1. A subgraph isomorphic to g_l = *context* $\cup$ *retraction* is identified in g.
2. If no isomorphic subgraph exists or if the isomorphic subgraph can be extended to match g_l plus the restriction (if one exists), r cannot be applied to g.
3. The elements of the subgraph isomorphic to the retraction are removed from g, leaving the host graph.
4. A graph isomorphic to the insertion is embedded into the host graph by the edges between the context and the insertion in r.

The application of a transformation rule is also conditioned by functional constraints. These constraints are related to the parameters of this rule, the attributes values of the graph nodes, etc. We propose to describe these transformation operations in Z language; each operation is expressed by the following schema:

Rule_name

$par_1?$; $par_2?$; ...; $par_n?$
$\Delta system\_name$

Pre − conditions :
functional constraints : formula on parameters, formulas on nodes attributes, etc.
structural constraints : formulas on the relations equivalent to the identification of g_l = *context* $\cup$ *retraction in the system graph*
Actions :
Operations on the sets of nodes/edges (insertion/retraction)

Where $Par_i?$ denote the inputs of the rule and Δ system denotes that the rule may change the system state.

The application of a transformation rule is then performed as follows: if the functional and structural constraints are satisfied, then the actions can be executed (insertion and/or retraction of the nodes and/or arcs).

3 Case study

To give more details on our approach, we present in this section a simple example. The Patient Monitoring system (*PMS*), which was used to illustrate works of [8] and [15]. To represent the communication architecture of this system, we chose the Producer/Consumer style.

For each service of the private clinic (pediatric, cardiology, maternity...) 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 patient 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. So, the nurse and the bed monitor behave respectively as a consumer component and a producer component.

3.1 System specification

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:

- The system must contain a maximum of 3 services.
- A service contains a maximum of 5 nurses and 15 patients.
- A patient must be always affected with only one service. This later must contain at least one nurse to take care of this patient.
- A nurse must be connected to only one service.

Hence, the specification of the system must consider the constraints of the Producer/Consumer style and the stated specific properties. In *Z* notation, we may combine the information contained in two schemas by including one in the declaration part of the other. The declarations are merged and the predicates conjoined. So, the specification is as follows: $[BED\_MONITOR, EV\_SER, NURSE]$

$$\boxed{\begin{array}{l} PMS1 \\ \hline Prod\_Cons[BED\_MONITOR, EV\_SER, NURSE] \end{array}}$$

$$\boxed{\begin{array}{l} PMS \\ \hline PMS1[PB/P, ES/S, CN/C] \\ NB\_CN : EV\_SER \rightarrow \mathbb{N} \\ NB\_PB : EV\_SER \rightarrow \mathbb{N} \\ \hline \#ES \leq 3 \\ \forall s : ES \bullet NB\_CNs \leq 5 \land NB\_PBs \leq 15 \\ \forall x : PB \bullet \\ \exists_1 s : ES \bullet (x, s) \in pushP \land (s, x) \in pullS \\ \land NB\_CNs \geqslant 1 \\ \forall s, z : ES;\ y : CN \bullet \\ (y, s) \in pullC \land (s, y) \in pushS \land (y, z) \in pullC \\ \land (z, y) \in pushS \Rightarrow s = z \end{array}}$$

Notice that *N_Producer*, *N_Service* and *N_Consumer* are instantiated respectively with *BED_MONITOR*, *EV_SER* and *NURSE* and we have renamed some variables of the schema $Prod\_Cons[N\_Producer, N\_Service, N\_Consumer]$ to facilitate their use; In *Z* notation, if *Schema* is a schema, then we write $Schema[new/old]$.

The function *NB_CN* computes for a given service the number of connected nurses and the function *NB_PB* computes for a given service the number of affected patients.

So, it is obvious that the graph depicted in figure 1 is one possible instance of *PMS* system.

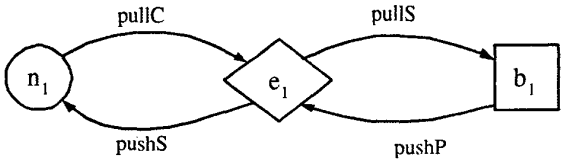

Figure 1. One possible configuration of the PMS system

3.2 Specification of evolution

In this section, we will present the specification of some rules allowing evolving our *PMS* system while taking into account the architectural properties.

- *Insertion of an event-service:* this rule inserts an instance of a component of type *EV_SER* provided that the system does not already contain three event services according to the first predicate in the *PMS*

schema. This rule is expressed by the following schema *insert_ES*:

insert_ES

$x? : EV\_SER$
ΔPMS

$\#ES < 3$
$ES' = ES \cup \{x?\}$

If we apply this rule to the graph of figure 1 with e_2 as parameter, we will obtain the graph given by figure 2.

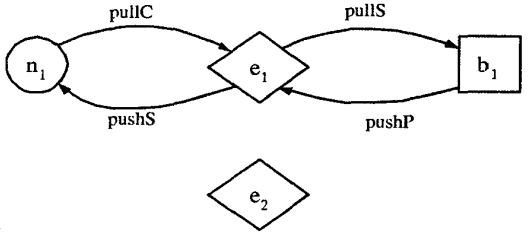

Figure 2. Graph obtained after the application of *insert_ES* rule to the graph of figure 1

- *Insertion of a patient:* the insertion of a new patient in the system consists in adding a new bed monitor. Thus, this rule inserts an instance of component of type *BED_MONITOR* and connects it to an event service under 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. Moreover, we should verify that this service does not already contain fifteen patients according to the second predicate in the *PMS* schema. This rule is expressed by the following schema *insert_PB*:

insert_PB

$z? : BED\_MONITOR$
ΔPMS
$x : EV\_SER$

$\exists y : CN \bullet x \in ES \wedge$
$(x, y) \in pushS \wedge (y, x) \in pullC \wedge$
$NB\_PBx < 15$
$PB' = PB \cup \{z?\}$
$pushP' = pushP \cup \{(z?, x)\}$
$pullS' = pullS \cup \{(x, z?)\}$
$NB\_PB'x = NB\_PBx + 1$

The application of this rule to the graph of figure 2 with b_2 as parameter generates the graph of figure 3.

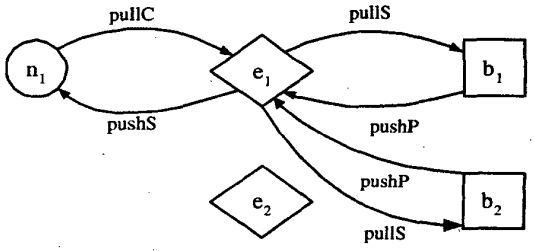

Figure 3. Graph obtained following the application of *insert_PB* to the graph of figure 2

- *Insertion of a nurse:* this rule inserts an instance of a component of type *NURSE* and connects it to an event service (that exists in the system) with the relation specified in the declaration part of *PMS* schema. In order to apply this rule, according to the second predicate of *PMS* schema, one has to check that the service in question does not already contain five nurses. We express this rule with the following schema *insert_CN*:

insert_CN

$y? : NURSE$
ΔPMS
$x : EV\_SER$

$x \in ES \wedge NB\_CNx < 5$
$CN' = CN \cup \{y?\}$
$pushS' = pushS \cup \{(x, y?)\}$
$pullC' = pullC \cup \{(y?, x)\}$
$NB\_CN'x = NB\_CNx + 1$

If we apply this rule to the graph of figure 3 with n_2 as parameter, there will be two possibilities : the nurse can be attached either to the service e_1 or to the service e_2. The choice is non-deterministic. We may obtain, for example, the graph given by figure 4.

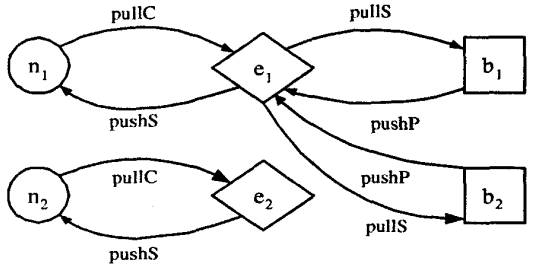

Figure 4. Graph obtained after the application of the rule *insert_CN* to the graph of figure 3

- Removal of a nurse: a nurse can leave the system if she is not responsible for any patient. Thus, this rule removes a component of type *NURSE* if and only if it is not connected to a service that contains patients and does not contain other nurses. We express this rule with the following schema *supp_CN*:

supp_CN

$y? : NURSE$
ΔPMS

$y? \in CN$
$\forall x : ES \bullet (y?, x) \notin pullC \wedge (x, y?) \notin pushS$
$CN' = CN \setminus \{y?\}$

If one tries to apply this rule, for example, to the graph of figure 4 with n_1 as parameter, one will notice that the second predicate of the rule is not satisfied. Consequently, the rule is, in this case, not applicable.

- *Removal of a patient:* when a patient leaves the system, the corresponding bed monitor is disconnected from the associated service and is removed from the system. This rule is expressed with the following schema *supp_PB*.

supp_PB

$z? : BED\_MONITOR$
ΔPMS
$x : EV\_SER$

$z? \in PB$
$x \in ES \wedge (x, z?) \in pullS \wedge (z?, x) \in pushP$
$PB' = PB \setminus \{z?\}$
$pushP' = pushP \setminus \{(z?, x)\}$
$pullS' = pullS \setminus \{(x, z?)\}$
$NB\_PB'x = NB\_PBx - 1$

- Disconnection of a nurse: a nurse can leave her service (without being removed from the system) only if this service does not contain patients or it contains other nurses who could deal with the patients. Thus, this rule makes it possible to disconnect a component of type *NURSE* only if one of these conditions is satisfied. We express this rule with the schema *disconnect_CN*.

disconnect_CN

$y? : NURSE$
ΔPMS
$x : EV\_SER$

$y? \in CN \wedge x \in ES \wedge$
$(x, y?) \in pushS \wedge (y?, x) \in pullC$
$NB\_CNx > 1 \vee NB\_PBx = 0$
$pushS' = pushS \setminus \{(x, y?)\}$
$pullC' = pullC \setminus \{(y?, x)\}$
$NB\_CN'x = NB\_CNx - 1$

- *Connection of a nurse:* It would have initially to be checked that the nurse is free, in other words, she is not attached to a service. Moreover, she could be connected to a service only if this last does not contain already five nurses in accordance with the second predicate of the system. This rule is specified by the schema *connect_CN*.

connect_CN

$y? : NURSE$
ΔPMS
$x : EV\_SER$

$y? \in CN$
$\forall z : ES \bullet (y?, z) \notin pullC \wedge (z, y?) \notin pushS$
$x \in ES \wedge NB\_CNx < 5$
$pushS' = pushS \cup \{(x, y?)\}$
$pullC' = pullC \cup \{(y?, x)\}$
$NB\_CN'x = NB\_CNx + 1$

In conclusion, we can show the relevance of graph grammars with respect to the way they handel software architectures.

3.3 Properties checking

In this section, we will show that the proposed specification preserves the consistency of the system and ensures its safety during its evolution. In order to achieve this, we have to prove the following theorem:

$$\forall c, c' \in Conf, \forall r \in Rules.(c \,\mathbf{sat} P) \wedge (c \xrightarrow{r} c') \Rightarrow c' \mathbf{sat} P$$

This theorem can be informally expressed as follows: c being a configuration satisfying the required properties P, and r being a rule which could be applied to c, then the new configuration c' obtained after the application of r on c, verify the required properties. For this purpose, it is sufficient to identify for each property the rules which may produce architectures violating the corresponding properties.

Taking as an example the property which indicates that a service contains a maximum of five nurses, the only rules

being able to violate it are those which connect a nurse to a service. The concerned rules are then *insert_CN* and *connect_CN*. However they have the following condition: $Nb\_CN(\_x) < 5$, which prevents them to be applied if the concerned service already contains five nurses. Then, the property will never be violated.

Among the *PMS* constraints, one of them stipulates that a patient must be always affected with a service. The only rules which may violate this property are those which manipulate connections of components of *BED_MONITOR* type with components of *EV_SER* type. In our case, it is about the *insert_PB* and *supp_PB* rules. The rule *insert_PB* allows to insert an instance of *BED_MONITOR* component and to connect it to a service simultaneously. Concerning the rule *supp_PB*, it removes completely a *BED_MONITOR* instance. Then, this constraint will never be violated.

We can also check the constraint which stipulates that the service to which the patient is assigned must contain at least one nurse. The risk lies in the rules being able to remove or disconnect a nurse or to connect a *BED_MONITOR* component to a service. The concerned rules are : *insert_PB*, *supp_CN* and *disconnect_CN*. The rule *insert_PB* assigns a bed monitor to a service only if this later contains at least one nurse. This does not violate the property. The rule *supp_CN* removes a nurse only if it is not connected to a service containing patients and no other nurses. Finally, the rule *disconnect_CN* disconnects a nurse from its service only if this later does not contain patients or if it contains at least another nurse. Thus, the property is, in fact, preserved.

4 Conclusion

We presented in this article 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, our 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.

Currently, we are using *Z/EVES* [10], a tool that supports the *Z* notation, for specifying software architectures and reasoning about their dynamics. In our future works, we plan to take into account the description of the behavioural aspect of software components. For this purpose, we will investigate the integration of a process algebra language (like CSP or CCS) into our framework.

References

[1] J. Dulay, N. Kramer, and J. Magee. Structuring parallel and distributed programs. *IEEE Software Engineering Journal*, 8(2):73–82, Mars 1993.

[2] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE transactions on Software Engineering*, 21(9):717–734, Septembre 1995.

[3] *Guide to the Rapide 1.0 Language Reference Manuals*. Rapide Design Team Program Analysis and Verification Group Computer Systems Lab Stanford University, 1997.

[4] R. Allen, R. Douence, and D. Garlan. Specifying and analysing dynamic software architectures. *Journal of Fundamental Approaches to Software Engineering*, 1382:21–37, 1998.

[5] C. Canal, E. Pimentel, and J.M. Troya. Specification and refinement of dynamic software architectures. *Journal of Software Architecture*, pages 107–125, Avril 1999.

[6] N. Aguirre and T. Maibaum. A temporal logic approach to component-based system specification and reasoning. In *5th ICSE Workshop on component-based software engineering*, Orlando, Florida, USA, Avril 2002.

[7] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, 1995.

[8] D. Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553, Juillet 1998.

[9] J. R. Abrial. *Programming as a mathematical exercise*. In C.A.R. Hoare, editor, Mathematical Logic and Programming Languages. Prentice-Hall International, 1985.

[10] Z/eves. http://www.ora.on.ca/Z-eves.

[11] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of z in isabelle/hol. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics — 9th International Conference*, LNCS 1125, pages 283–298. Springer Verlag, 1996.

[12] Appligraph. applications of graph transformations. Technical Report 22565, Hans-jörg Kreowski and Detlef plump editors, Mai 2001.

[13] M.A. Wermelinger. *Specification of software architecture reconfiguration*. PhD thesis, Université Nova de Lisbon, Septembre 1999.

[14] S. M. Kaplan, J. P. Loyall, and S. K. Goering. Specifying concurrent languages and systems with δ-grammars. *Research Directions in Concurrent Object-Oriented Programming*, 1993.

[15] I. Warren and I. Sommerville. Dynamic configuration abstraction. In W. Schäfer and P. Botella, editors, *Proceedings of the Fifth European Software Engineering Conference*, pages 173–190. Springer-Verlag, 1995.