Consistent Reconfiguration for Publish/Subscribe Architecture Styles

Imen Loulou, Ahmed Hadj Kacem, Mohamed Jmaiel
University of Sfax
Laboratory LARIS
B.P. 1088, 3018 Sfax, Tunisia
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
Khalil@laas.fr

Abstract

This paper addresses the dynamic aspect of Publish/Subscribe architectures. We propose an integrated formal approach for the specification and verification of reconfigurations over the topology of Publish/Subscribe architecture styles. We integrate a functional approach and a structural approach based on graph grammars. The purpose is to express dynamism while offering a simple specification which can be easy to read and to understand. We use the specification language Z to describe the constraints made on the architecture style and we describe the dynamic of architecture via guarded graph-rewriting rules whose guards mainly describe the pre- and post-conditions ensuring in this way the preservation of stylistic constraints. To ensure that the system is evolving correctly, we elaborate a verification process which validates each rule, whose semantic is described in Z notation, using the Z-EVES theorem prover.

Keywords: software architecture, Publish/Subscribe architectural style, dynamic architecture, graph rewriting, formal specification.

1. INTRODUCTION

The publish/subscribe component model is an emerging paradigm to support distributed systems composed of highly evolvable and dynamic federations of components [1, 2, 3, 4]. According to this paradigm, events are propagated in an anonymous way between the components that have generated them, called producers, and the receivers, called consumers. This transparency is due to the event-service which is the glue that ties together distributed components in Publish/Subscribe architecture. It establishes a loosely coupled communication since entities do not keep references to one another and do not communicate directly [5]. A major issue is to preserve the system consistency during reconfiguration. The system must be left in a *correct* state after reconfiguration, by maintaining the conformity of its new architecture with respect to a set of structural properties or architecture style.

In our previous work [6], we proposed a compositional formal approach to design correct publish/subscribe architecture styles. We provided a reusable set of patterns and the corresponding composition rules to build architecture styles. Patterns were described as graphs and we used the Z notation to specify formally the semantic of each pattern and each rule. We proposed also, to construct the Z specification of the designed architecture style based on applied rules and we proved consistency and correctness using the Z-Eves theorem prover.

In this paper, we address the dynamic aspect of Publish/Subscribe architectures. 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. We describe

reconfiguration operations via a new notation based on graph rewriting rules and Z notation. This integration offers to software architects an intuitive and formal way to specify the dynamic aspect of software architecture based on visual notation. We propose also to specify the semantic of each reconfiguring operation 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. We use *Z/EVES* theorem prover to check that all reconfiguring operations respect the defined architecture style. Our approach has the advantage of providing a unified framework to formally specify both static and dynamic aspects of a Publish/Subscribe architecture. This ensures a coherence between the two aspects.

The remainder of this paper is organized as follows: in section 2, we present a survey of related works, in section 3, we introduce the main characteristics of a Publish/Subscribe architecture style. Then, we present our proposed approach to describe the reconfiguration and we explain the verification process to validate the reconfiguring operations. Section 4 presents a case study which illustrates our approach. It also explains the verification proofs to be performed. In section 5, we give the concluding remarks and future work perspectives.

2. RELATED WORKS

This survey will focus on research done in the area of software architecture description, on one hand. On the other hand, it will concentrate on some works which address the dynamic aspect of publish/subscribe systems.

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 [7], Rapid [8, 9], Wright [10] and LEDA [11]. They provide modelling tools which help the designers to structure a system and to compose its elements. Generally, ADLs allow 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 rigorous analysis and verification of architectural properties. The second class makes use of formal methods for specifying and verifying software architectures. We noted that 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 [12] for its high expressive power and formal reasoning. Nevertheless, reconfiguration is not well exhibited with this formalism. In [13], 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, [14] 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. These rules explain clearly the topological changes, but they don't allow to express post-conditions or certain logical conditions such as the absence of a communication link between two software components.

Recent research addresses the dynamic reconfiguration of Publish/Subscribe systems. However, the major issues which have been treated are about notification and (un)subscription loss as well as duplication during reconfiguration [15, 16, 17]. In addition, most of them [15, 18, 19, 20] consider reconfiguration by repeatedly exchanging a single link in the network of dispatchers. In [15] for instance, algorithms were presented that prevent the loss or duplication of notifications and maintain the message ordering. In [18], authors present a description and analysis of a novel algorithm to deal with the reconfiguration of the dispatching infrastructure. The strength of this

algorithm is its ability to minimize the portion of the system affected by the reconfiguration by exploiting a novel concept they refer to as the reconfiguration path.

3. OUR APPROACH

In this section, we begin by introducing the main features of a Publish/Subscribe architecture style. Then, we present our approach to describe architecture reconfiguration and show how we take into account these features. Finally, we explain the verification process which we elaborated to validate the reconfiguring operations with respect to the architecture style.

3.1. Publish/Subscribe architecture style

In the literature [21, 22, 23, 24, 25, 2], several event-based architectures have been proposed. We distinguish three event models. The peer to peer model allows consuming entities to subscribe at specific named producing entities directly and producing entities to deliver events to specific named subscribed entities directly. The Java distributed event model [26] serves as an example of a peer to peer event model. The implicit event model lets consuming entities subscribe to particular event types rather than at an explicitly addressed entity or mediator. The producing entities generate events of a given type, which are then delivered to the subscribed entities. The Cambridge event model (CEM) [27] is based on an implicit event model. The third event model using a mediator allows the consuming entities to subscribe at an indicated mediator (event dispatcher) and the producing entities to deliver events to the mediator, which then forwards them to the subscribed entities. In this paper, we are interested in mediator-based event models. We distinguish architectures using a centralized event service and architectures using a distributed event service. In the first case, a single dispatcher is used and in the second case the event service uses a network of event dispatchers that provide access points to clients. Dispatchers cooperate together to route information from the producers to the subscribed entities. This interaction is governed by a principle of information propagation requiring that produced information have to reach all subscribed consumers. To guarantee this property in the case of a distributed event service, we make two constraints:

- C1: all dispatchers have to be interconnected.
- C2: if two dispatchers communicate together then it is necessary that the information coming from the one reaches the other. So, the communication has to be bidirectional.

In addition, several interconnection topologies have been proposed: hierarchical, acyclic peer-to-peer, and general peer-to-peer [23]. The hierarchical topology is an extension of the centralized approach in which a set of interconnected dispatchers maintain a master/client relationship with other dispatchers. In the acyclic peer-to-peer topology, dispatchers communicate with each other symmetrically as peers, adopting a protocol that allows a bidirectional flow of subscriptions, advertisements, and notifications. In this topology, any two dispatchers are connected with at most one path. Removing the constraint of acyclicity from the acyclic peer-to-peer topology, we obtain the general peer-to-peer topology.

Another characteristic is worth a particular attention: the way subscribers can express the events they are interested in (usually called subscription schema). The existing literature distinguishes among topic-based, content-based and type-based systems [2, 21]. The topic-based subscription schema is based on the notion of *topic* or *subject*, a concept used to group events published in the system in not necessarily disjoint sets. Publishers generate events belonging to one or more topics, while subscribers express interest in some of these topics and will therefore receive notifications about events published in that particular topic. The content-based (or property-based [28]) publish/subscribe variant improves on topics by introducing a subscription schema based on the actual content of the considered events. In other terms, events are not classified according to some predefined external criterion (e.g., topic name), but according to the properties of the events themselves. The type-based subscription schema strongly resembles the topic-based approach with the notion of event *kind* (i.e. subject) directly matched with that of event *type* [29]. This enables a closer integration of the language and the middleware.

3.2. The dynamic Aspect

The four fundamental reconfiguration operations, provided by almost all languages and systems, are creation and removal of components and connections [30]. Graph grammars showed their relevance to express these operations of reconfiguration in particular in the works described in [30] and [14]. A graph production rewrites a graph into another graph, deleting some elements (nodes and edges), generating new ones and preserving others. The study that we carried out on the graph rewriting techniques enabled us to appreciate the transformation principle of the Δ notation proposed in [31]. This notation expresses the structural constraints (existence/absence of a motif) via four sections:

- *Retraction* : the fragment of the graph removed during the rewrite.
- *Insertion* : the fragment that is created and embedded into the graph during the rewrite.
- *Context* : the fragment that is identified but not changed during the rewrite.
- *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

However, in publish/subscribe architecture styles, we need also to specify constraints on some attributes values related to the subscription model. Furthermore, we need to define the post-conditions of an operation to describe what must be ensured after the application of an operation. Indeed, any operation defined to reconfigure the network of dispatchers, for instance, has to ensure at least the constraints *C1* and *C2* (described in section 3.1) after its application. In addition, sometimes it is easier to express some structural restrictions functionally (called Negative Application Condition (NAC) from now on) rather than visually. So, we propose a new notation which contains a structural part following the Δ notation, and an analytic part specified in Z notation. Our notation is depicted in figure 1. So, given a rule r , it is applied to a graph g by the following steps:

1. A subgraph isomorphic to $g_l = \text{context} \cup \text{retraction}$ 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 *NAC*, the Application condition (*AC*) and the post-conditions parts are evaluated. They must be satisfied otherwise r cannot be applied.
4. The elements of the subgraph isomorphic to the retraction are removed from g , leaving the host graph.
5. A graph isomorphic to the insertion is embedded into the host graph by the edges between the context and the insertion in r .

FIGURE 1: The proposed notation

3.2.1. Formal semantic

Besides the fact that our notation fills the identified limits, the partial introduction of the formal description constitutes a preparation for a total formal semantic translation in a purpose of analysis

and verification. So, we propose to describe the semantic of each reconfiguring operation via a schema operation Z as follows:

<i>Rule_Name</i>
$\Delta system_name$
$in_1?; in_2?; \dots; in_n?$
<i>NAC + Restriction :</i> <i>structural and functional constraints</i>
<i>Application conditions :</i> <i>functional constraints</i> <i>structural constraints</i>
<i>Post – conditions</i>
<i>Reconfiguration :</i> <i>(Insertion / Retraction)</i>

Where:

- $in_i?$ are the input parameters of the operation,
- $\Delta system_name$ denotes that the rule will change the system state.
- The functional constraints are about the attributes values, the number of component instances, etc.

3.2.2. Correctness preservation during evolution

In order to verify that a given reconfiguring operation makes evolve the system correctly, we propose to use the precondition theorem provided in Z notation. This theorem can identify the necessary pre- and post-conditions for an operation to preserve the properties defined by the architecture style. This theorem is described as follows:

Theorem correct_preconditions

$\forall State; in? : INPUT \mid precondition(State, in?)$
 • pre Op

Where:

- *State* is the schema name of the designed style,
- $in?$ the input parameters of the operation,
- $precondition(State, in?)$ are the preconditions upon state variables and input parameters which must be satisfied beforehand,
- *Op* is the schema operation name,
- and pre Op characterizes the collection of before states and inputs for which some after state can be shown to exist. Formally:

$$pre Op = \exists State'; out! : OUT \bullet Op$$

So, the expanded theorem is described as follows:

Theorem correct_preconditions

$\forall State; in? : INPUT \mid precondition(State, in?)$
 • $\exists State'; out! : OUT \bullet Op$

This theorem allows to verify the following: what ever could be the system state, with the input parameters given in the schema operation and the defined pre- and post-conditions, a *correct after state* can be shown to exist.

The first step in the verification process is to prove this theorem for each reconfiguring operation. Once proved, we can check if the preconditions identified by this theorem and those described in the schema operation are the same. If they tally, then we can say that the operation is valid.

Furthermore, we propose to use the same theorem to correct invalid operations. Indeed, we can complete or replace the pre- and post-conditions described in the schema operation by those which are identified by the theorem.

4. CASE STUDY

To illustrate our approach, we choose a case study in the context of emergency operation activities occurring during a crisis situation and introduced in NETQoS Project [32]. The application involves structured groups of robots or military personnel which cooperate for the realization of a common mission. The activity is divided into two successive steps: an exploration step of the investigating field and an action step following the discovery of a critical situation. A team of emergency intervention is constituted of participants having different jobs: a controller, coordinators and investigators (max 10 investigators). In the exploration step, investigators provide continuous descriptive feedbacks (D_Inv) and periodical Produced feedbacks (P_Inv), expressing the analysis of the situation, to their responsible coordinators. The coordinators complete the received analysis to send them (P_Coord) thereafter to the controller. Having discovered a critical situation, the concerned investigator keeps the same functions as in step 1 but provides feedbacks D_Inv to the other investigators. Now, the other investigators report only feedbacks P_Inv to controllers on the basis of feedbacks D_Inv transmitted by the investigator who has discovered the critical situation.

So, both investigators and coordinators are producer/consumer components and the controller is a consumer entity. The propagation of events between these participants is mediated by a network of event dispatchers called managers. This interaction is governed by the principle of information propagation described by the constraints C1 and C2 mentioned in section 3.1. We have to notice also that managers communicate together using the push mode to propagate events.

We assume also that the subscription model is topic-based. Publishers generate events belonging to one or more topics (D_Inv for example), while subscribers express interest in some of these topics and will therefore receive notifications about events published in that particular topic.

So the corresponding Publish/Subscribe architecture style is depicted in figure 2 and the formal specification is as follows:

[TOPIC]

<p><i>Emergency</i></p> <p>$Cont : \mathbb{F} \text{ Controller}$ $M : \mathbb{F} \text{ Manager}$ $Inv : \mathbb{F} \text{ Investigator}$ $Coord : \mathbb{F} \text{ Coordinator}$ $Push_M : \text{Manager} \leftrightarrow \text{Controller}$ $Push_IM : \text{Investigator} \leftrightarrow \text{Manager}$ $Push_MI : \text{Manager} \leftrightarrow \text{Investigator}$ $Push_MC : \text{Manager} \leftrightarrow \text{Coordinator}$ $Push_CM : \text{Coordinator} \leftrightarrow \text{Manager}$ $PushDD : \text{Manager} \leftrightarrow \text{Manager}$</p> <hr/> <p>$\#Inv \leq 10$ $\forall m1, m2 : M$ <ul style="list-style-type: none"> $((m1, m2) \in PushDD \Rightarrow (m2, m1) \in PushDD)$ $\forall x, y : M; i : \mathbb{N} \mid x \neq y$ <ul style="list-style-type: none"> $\exists T : \text{seq EvDispatcher}$ <ul style="list-style-type: none"> $i + 1 \leq \#T$ $i \geq 1$ $\text{ran } T \subseteq M$ $\wedge (Ti, T(i+1)) \in PushDD$ $\wedge T1 = x$ $\wedge T(\#T) = y$ $\forall x : Cont$ <ul style="list-style-type: none"> $x.topic_cons = \{P_Coord\}$ $\forall y : Inv$ <ul style="list-style-type: none"> $y.topic_prod = \{D_Inv, P_Inv\}$ $\wedge y.topic_cons = \{D_Inv\}$ $\forall z : Coord$ <ul style="list-style-type: none"> $z.topic_prod = \{P_Coord\}$ $\wedge z.topic_cons = \{D_Inv, P_Inv\}$ $\forall w : M$ <ul style="list-style-type: none"> $w.topics = \{D_Inv, P_Inv, P_Coord\}$ </p>

Where P_Coord , D_Inv and P_Inv are the topics of interest and they are declared as follows:

<p>$D_Inv, P_Inv, P_Coord : TOPIC$</p> <hr/> <p>$D_Inv \neq P_Inv \wedge$ $D_Inv \neq P_Coord \wedge P_Inv \neq P_Coord$</p>

FIGURE 2: Emergency Operations scenario

4.1. Reconfiguration Specification

In this section, we present the specification of some rules allowing evolving our *Emergency* system while taking into account its defined properties.

- *Insertion of an investigator*: this rule inserts an instance of a component of type *Investigator* and connect it to a *manager*, provided that the investigator to be inserted supports the topics defined by the fifth predicate in the *Emergency* schema. In addition, the number of investigators in the architecture to be reconfigured must be less of 10 according to the first predicate. This rule is visually expressed by figure 3 and the semantic is described by the schema *Insert_Investigator*.

FIGURE 3: Insertion of an investigator

<i>Insert_Investigator</i> Δ <i>Emergency_refined_subscription</i> <i>inv2?</i> : <i>Investigator</i> <i>m1?</i> : <i>Manager</i>
<i>inv2?</i> $\notin$ <i>Inv</i> $\#Inv < 10$ <i>inv2?.topic_prod</i> = { <i>D_Inv</i> , <i>P_Inv</i> } <i>inv2?.topic_cons</i> = { <i>D_Inv</i> } <i>m1?</i> $\in$ <i>M</i> <i>Inv'</i> = <i>Inv</i> $\cup$ { <i>inv2?</i> } <i>Push_IM'</i> = <i>Push_IM</i> $\cup$ {(<i>inv2?</i> , <i>m1?</i>)} <i>Push_MI'</i> = <i>Push_MI</i> $\cup$ {(<i>m1?</i> , <i>inv2?</i>)}

- *Link exchange*: An existing link between two managers is removed from the topology and a new link is added instead. In this case, we have to ensure that once this operation is applied the topology of managers remains interconnected according to the constraint C1 described by the third predicate of the schema *Emergency*. In addition, we have to ensure that the connection/disconnection respect the constraint C2 described by the second predicate. These post-conditions are described in the analytic part of this rule which is expressed by the figure 4 and which semantic is described by the schema *Link_Exchange*.

$$\begin{aligned}
 & \forall mg1, mg2 : M \\
 & \bullet ((mg1, mg2) \in PushDD \setminus \{(m2?, m3?), (m3?, m2?)\} \cup \{(m1?, m3?), (m3?, m1?)\}) \\
 & \Rightarrow (mg2, mg1) \in PushDD \setminus \{(m2?, m3?), (m3?, m2?)\} \cup \{(m1?, m3?), (m3?, m1?)\}) \\
 & \forall x, y : M; i : \mathbb{N} \mid x \neq y \\
 & \bullet \exists T : seq EvDispatcher \\
 & \bullet i + 1 \leq \#T \wedge i \geq 1 \wedge ran T \subseteq M \\
 & \wedge (Ti, T(i+1)) \in PushDD \setminus \{(m2?, m3?), (m3?, m2?)\} \cup \{(m1?, m3?), (m3?, m1?)\} \\
 & \wedge T1 = x \wedge T(\#T) = y
 \end{aligned}$$

FIGURE 4: Link exchange

$Link_Exchange$ $\Delta Emergency_refined_subscription$ $m1?, m2?, m3? : Manager$
$m1? \in M \wedge m2? \in M \wedge m3? \in M$ $(m2?, m3?) \in PushDD \wedge (m3?, m2?) \in PushDD$ $\forall mg1, mg2 : M$ <ul style="list-style-type: none"> $((mg1, mg2) \in PushDD \setminus \{(m2?, m3?), (m3?, m2?)\} \cup \{(m1?, m3?), (m3?, m1?)\})$ $\Rightarrow (mg2, mg1) \in PushDD \setminus \{(m2?, m3?), (m3?, m2?)\} \cup \{(m1?, m3?), (m3?, m1?)\})$ $\forall x, y : M; i : \mathbb{N} \mid x \neq y$ <ul style="list-style-type: none"> $\exists T : seq EvDispatcher$ $i + 1 \leq \#T \wedge i \geq 1 \wedge ran T \subseteq M$ $\wedge (Ti, T(i+1)) \in PushDD \setminus \{(m2?, m3?), (m3?, m2?)\} \cup \{(m1?, m3?), (m3?, m1?)\}$ $\wedge T1 = x \wedge T(\#T) = y$ $PushDD' = PushDD \setminus \{(m2?, m3?), (m3?, m2?)\} \cup \{(m1?, m3?), (m3?, m1?)\}$

4.2. Consistency checking

In this section, we will show how we use the theorem prover of Z-Eves to formally verify for each operation that the proposed pre-condition is in fact the required pre-condition to ensure the consistency of the system and its safety during its evolution.

4.2.1. Validation of Insert_Investigator

As it is explained in section 3.2.2, the first step of the verification process is to identify the necessary pre- and post-conditions of an operation. So, for the *Insert_Investigator* operation, we have to prove the following theorem:

Theorem precondition_invest

$$\begin{aligned}
&\forall \text{Emergency}; m1? : \text{Manager}; inv2? : \text{Investigator} \\
&\quad | inv2? \notin Inv \\
&\quad \wedge \#Inv < 10 \\
&\quad \wedge inv2?.topic_prod = \{D_Inv, P_Inv\} \\
&\quad \wedge inv2?.topic_cons = \{D_Inv\} \\
&\quad \wedge m1? \in M \\
&\bullet \text{pre Insert_Investigator}
\end{aligned}$$

However, this theorem could not be proved since the pre-conditions described in the schema operation are not sufficient. After some iterations with the theorem prover, we succeeded in proving this theorem:

Theorem precondition_invest

$$\begin{aligned}
&\forall \text{Emergency}; m1? : \text{Manager}; inv2? : \text{Investigator} \\
&\quad | inv2? \notin Inv \\
&\quad \wedge \#Inv < 10 \\
&\quad \wedge \#(Inv \cup \{inv2?\}) \leq 10 \\
&\quad \wedge inv2?.topic_prod = \{D_Inv, P_Inv\} \\
&\quad \wedge inv2?.topic_cons = \{D_Inv\} \\
&\quad \wedge m1? \in M \\
&\quad \wedge (\forall x : Inv \cup \{inv2?\} \\
&\quad \quad \bullet (x.topic_prod = \{D_Inv, P_Inv\} \\
&\quad \quad \wedge x.topic_cons = \{D_Inv\})) \\
&\bullet \text{pre Insert_Investigator}
\end{aligned}$$

As we can notice, the theorem requires post-conditions which are not specified in the schema operation. In fact, the principle of this theorem is to ensure the conformity of the *after state* with respect to the architecture style. So, since the set of investigators has been changed (Inv'), the theorem verifies all the properties related to this set and described in the schema style *Emergency*. Whence the post-condition related to the cardinality of this set and the post-condition related to the topics supported by the investigators.

In this case, the second step of the verification process is to correct the schema operation *Insert_Investigator* by adding the identified post-conditions, so that we can confirm that this reconfiguring operation leaves the system in a correct state.

4.2.2. Validation of Link_Exchange

We proceeded in the same way to verify if the operation *Link_Exchange* preserves the stylistic constraints. The theorem was successfully proved from the first iteration without adding anything. So the schema operation *Link_Exchange* describes all necessary pre- and post-conditions to leave the system *Emergency* in a correct state.

5. CONCLUSION

We proposed in this paper a formal approach for the specification of dynamic architectures of Publish/Subscribe systems. This approach is based on an integration of graph-based semantics in the framework of the Z formal language. 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. We elaborate a verification process using the Z/EVES theorem prover to check that all defined operations preserve the consistency of the architecture style during its evolution. This process is also used to correct the inconsistent operations by adding the necessary pre- and post-conditions to leave the system in a correct state.

In our future works, we plan to develop a plugin into Eclipse allowing software architects to design visually, using an extended-UML graphical notation, correct by design Architecture Styles and

the corresponding reconfiguration operations. The proposed functions include also generating automatically valid and correct Z specification.

REFERENCES

- [1] L. Baresi, C. Ghezzi, and L. Zanolin. Modeling and validation of publish/subscribe architectures. *S. Beydeda and V. Gruhn, editors, Testing Commercial off the shelf Components And Systems*, pages 273–292, 2005.
- [2] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
- [3] René Meier and Vinny Cahill. Taxonomy of distributed event-based programming systems. *Comput. J.*, 48(5):602–626, 2005.
- [4] José Luiz Fiadeiro and Antónia Lopes. A formal approach to event-based architectures. In *FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006*, volume 3922 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2006.
- [5] Alberto B. Raposo, Adailton J.A. da Cruz, Christian M. Adriano, and Léo P. Magalhães. Coordination components for collaborative virtual environments. *Computers & Graphics*, 25:1025–1039, 2001.
- [6] I.Loulou, A.H.Kacem, M.Jmaiel, and K.Drira. A formally specified framework for elaborating event-based architectural styles correct by design. In *The 9th Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI'06)*, Agadir-Morocco, December 2006.
- [7] J. Dulay, N. Kramer, and J. Magee. Structuring parallel and distributed programs. *IEEE Software Engineering Journal*, 8(2):73–82, Mars 1993.
- [8] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
- [9] *Guide to the Rapide 1.0 Language Reference Manuals*. Rapide Design Team Program Analysis and Verification Group Computer Systems Lab Stanford University, 1997.
- [10] R. Allen, R. Douence, and D. Garlan. Specifying and analysing dynamic software architectures. *Journal of Fundamental Approaches to Software Engineering*, 1382:21–37, 1998.
- [11] C. Canal, E. Pimentel, and J.M. Troya. Specification and refinement of dynamic software architectures. *Journal of Software Architecture*, pages 107–125, Avril 1999.
- [12] 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.
- [13] 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.
- [14] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions On Software Engineering*, 24(7):521–533, 1998.
- [15] Helge Parzyjegla, Gero Gero Muhl, and Michael A. Jaeger. Reconfiguring publish/subscribe overlay topologies. In *ICDCSW '06: Proceedings of the 26th IEEE International Conference Workshops on Distributed Computing Systems*, page 29, Washington, DC, USA, 2006. IEEE Computer Society.
- [16] Michael A. Jaeger, Gero Mühl, Matthias Werner, and Helge Parzyjegla. Reconfiguring self-stabilizing publish/subscribe systems. In Radu State, Sven van Meer, Declan O'Sullivan, and Tom Pfeifer, editors, *17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2006)*, volume 4269 of *LNCS*, pages 233–238, Dublin, Ireland, oct 2006. Springer.
- [17] Bogumil Zieba and Marten van Sinderen. Preservation of correctness during system reconfiguration in data distribution service for real-time systems (dds). In *ICDCSW '06: Proceedings of the 26th IEEE International Conference Workshops on Distributed Computing Systems*, page 30, Washington, DC, USA, 2006. IEEE Computer Society.

- [18] Gianpaolo Cugola, Davide Frey, Amy L. Murphy, and Gian Pietro Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1134–1140, New York, NY, USA, 2004. ACM Press.
- [19] G. Cugola, G. P. Picco, and A. L. Murphy. Towards dynamic reconfiguration of distributed publish-subscribe middleware. In *the 3rd International workshop on Software Engineering and Middleware (SEM02), co-located with the 24th International Conference on Software Engineering (ICSE02)*, volume 2596, pages 187–202, Orlando (FL), USA, May 2002. Lecture Notes in Computer Science.
- [20] G. P. Picco, G. Cugola, and A. Murphy. Efficient content-based event dispatching in the presence of topological reconfigurations. In *the 23 International Conference on Distributed Computing Systems (ICDCS)*, 2003.
- [21] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.
- [22] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *International Conference on Distributed Computing Systems, Workshops (ICDCSW '02)*, pages 611–618, Vienna, Austria, July 2002. IEEE Computer Society.
- [23] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.
- [24] René Meier. State of the art review of distributed event models. Technical report, Department of Computer Science, University of Dublin, Trinity College, 2000.
- [25] Rene Meier and Vinny Cahill. Taxonomy of distributed event-based programming systems. In J. Bacon, L. Fiege, R. Guerraoui, A. Jacobsen, and G. Mühl, editors, *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, 2002.
- [26] Sun Microsystems Inc. Java distributed event specification: Sun microsystems inc. Technical report, 1998.
- [27] Jean Bacon, John Bates, Richard Hayton, and Ken Moody. Using events to build distributed applications. In *SDNE '95: Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments*, page 148, Washington, DC, USA, 1995. IEEE Computer Society.
- [28] D. ROSENBLUM and A. WOLF. A design framework for internet-scale event observation and notification. In *6th European Software Engineering Conference/ACM SIGSOFT 5th Symposium on the Foundations of Software Engineering*, page 344360, New York, NY, 1997. ACM Press.
- [29] P. EUGSTER, R. GUERRAOUI, and C. DAMM. On objects and events. In *OOPSLA'01 Conference on Object Oriented Programming Systems Languages and Applications.*, page 254269, New York, NY, 2001. ACM Press.
- [30] M.A. Wermelinger. *Specification of software architecture reconfiguration*. PhD thesis, Université Nova de Lisbon, Septembre 1999.
- [31] 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.
- [32] Deliverable d2.1: Qos solution analysis and user/application requirements and business scenarios. Technical Report IST-2005-033516-WP2-D2.1-V1.0, NETQoS Project, 2006.