Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 215 (2008) 5–21
www.elsevier.com/locate/entcs

Using Aspects for Enforcing Formal Architectural Invariants

Slim Kallel[1]

Software Technology Group
Darmstadt University of Technology
Darmstadt, Germany

Anis Charfi

SAP Research CEC Darmstadt
Darmstadt, Germany

Mohamed Jmaiel

ReDCAD Laboratory
National Engineering School of Sfax
Sfax, Tunisia

Abstract

Formal methods such as Z and Petri nets can be used to specify invariants that should hold during the execution of component-based applications such as those regarding changes in the architecture of the application and valid sequences of architecture reconfigurations. Integrating logic for checking and enforcing these invariants into the application's implementation is generally done by adding appropriate code to the functional application code. In this paper, we discuss several limitations of this approach that may ensue in a disconnection between the application implementation and its formal specification.
We propose an approach for specifying and enforcing architectural constraints, which combines formal methods and Aspect-Oriented Programming. We use the Z notation for describing the architectural invariants of the application and Petri nets for modeling coordination protocols. At the implementation level, aspects intercept architecture reconfiguration events and check according to the formal specification and the coordination protocol whether a reconfiguration action can be performed.

Keywords: Software architecture, Z notation, Petri nets, Aspect-Oriented Programming

[1] Email: kallel@st.informatik.tu-darmstadt.de

doi:10.1016/j.entcs.2008.06.018

1 Introduction

The software applications of today's organizations consist generally of several distributed software components. Such applications are characterized by a dynamic architecture, which evolves over the time in order to respond to the user requirements. For instance, new components may be added and existing connections between the components may be modified during execution. When such reconfigurations are done it is necessary to ensure that no faults are caused and that the software application works correctly.

To guarantee the reliability and consistency of the architectural evolution of distributed component-based applications, we propose using formal specifications. In this way, one could define the architectural constraints and coordination protocols that must be fulfilled by each reconfiguration of the application. Integrating logic for checking and enforcing architectural constraints and coordination protocols into the application's implementation is generally done by adding appropriate control code to the functional application code, as shown in [33,26]. These approaches provide the necessary control functionality, but they exhibit several limitations, which may ensue in a disconnection between the application implementation and its formal specification.

First, the control code that implements the constraints is written manually in these approaches. Second, this code is not well-modularized as it is tangled with the functional code of the application and scattered across the implementation of different components, which makes it non reusable. Third, the code that implements the constraints may not be conform to the formal specification. This is accentuated especially by the scattering problem. Fourth, if the formal specification changes, it is necessary to change the code that implements the constraints manually.

To solve these problems, we propose a novel approach for the runtime verification of distributed applications, which combines formal methods and Aspect-Oriented Programming (AOP) [17]. This approach covers the static structural aspect (i.e., specification of components, connections, and constraints) , the dynamic aspect (i.e., specification of the reconfiguration operations and their preconditions), and the coordination aspect (i.e., specification of the execution order of reconfiguration operations) of the software architecture. It fosters an organization of distributed component-based software systems in three phases: the formal specification phase, the base code implementation phase, and the aspect code implementation phase. In the first phase, the user specifies the constraints that should be fulfilled when the application evolves: architectural constraints are specified using the Z notation and coordination protocols are specified using Petri nets. In the base code implementation phase, the user writes the functional code of the different

components (here we use Java). This code does not contain any control logic. In the aspect code implementation phase, we use AspectJ aspects, which intercept reconfiguration events and check according to the formal specification whether the reconfiguration events can be performed.

This approach yields several benefits. It enables a more reliable control of the architectural evolution of component-based applications as it is based on formal methods. Moreover, using an aspect-based module to control architecture reconfiguration operations, mismatches between the implementation of the application and its formal specification are unlikely. In addition, the control code of the component-based distributed application becomes more reusable as it is well-modularized in aspects.

The remainder of this paper is organized as follows. In Section 2, we introduce the Z specification language, Petri nets, and Aspect-Oriented Programming. In Section 3, we present our approach for controlling the architecture evolution of component-based applications. Section 4 describes our case study collaborative authoring system. Section 5 reports on some related work and section 6 concludes this paper.

2 Background

In this section, we make use of two well-known formal methods, namely Z notation and Petri nets, for specifying respectively architectural constraints and coordination protocols. In addition, we use Aspect-Oriented Programming for modularizing the control and coordination code.

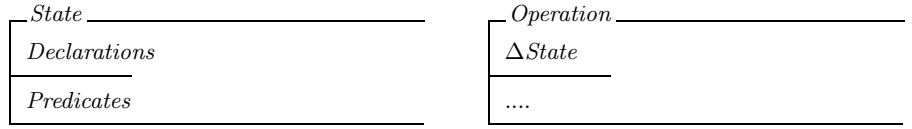
2.1 Z specification language

The Z notation, as presented in [30], is a formal specification language. Z defines a mathematical language, a schema language, and a refinement theory between abstract data types. The mathematical language is based on the set theory and on mathematical logic i.e., first order predicate logic. The schema language allows to describe the state of a system and the manners according to which this state can change. The refinement theory allows to develop a system by building an abstract model from a system design.

A Z specification can be defined as a collection of state schemes and operation schemes. The state schema *State* describes the system state and the invariant relationships, which are maintained when the system is updated. This schema consists of two parts: a declaration part and a predicate part.

The operation schemes *Operation* define the possible operations in the system, the relationship between their inputs and outputs, and the state changes resulting from their execution. The operation schema comprises the state

State before and the state *State'* after performing the operation. These two states are represented in the schema language by the $\Delta State$.



2.2 Petri nets

Petri nets [22] are a graphical and mathematical tool to model and analyze discrete systems. In Petri nets, the different states of a system are modeled by *places* and *tokens*. The events are represented by *transitions* between places. Formally, a Petri net can be defined as a 5-tuple $< P, T, F, W, M_0 >$, where: $P = \{p_1, ...p_m\}$ is a finite set of places; $T = \{t_1, ...t_n\}$ is a finite set of transitions with $(P \cap T) = \varnothing$ and $(P \cup T) \neq \varnothing$; $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs; $W : F \rightarrow \mathbb{N}_1$ is a weight function and $M_0 : P \rightarrow \mathbb{N}$ is an initial marking where for each place $p \in P$ there are $n \in \mathbb{N}$ tokens.

The system behavior can be described in terms of the system state and its changes. To simulate the dynamic behavior of the system, the state or marking will be changed according to the following rules:

- A transition t is enabled, if each input place p_i is marked with at least $W(p_i, t)$ tokens.

$$\forall\, p_i \in {}^\bullet t, M(p_i) \geq W(p_i, t). \qquad \text{[R1]}$$

- If a transition t is enabled for the marking M then the enabling of t will lead to the new marking M':

$$\forall\, p_i \in {}^\bullet t, M'(p_i) = M(p_i) - W(p_i, t) + W(t, p_i) \qquad \text{[R2]}$$

where $W(P_i, t)$ is the weight of the arc (P_i, t); $W(t, P_i)$ is the weight of the arc (t, P_i) and ${}^\bullet t$ is the set of input places of the transition t.

2.3 Aspect-Oriented Programming

Aspect-Oriented Programming [17] is a programming paradigm that allows the modularization of concerns that cut across the implementation of a software application, such as logging, persistence, and security.

According the *separation of concerns* principle, AOP provides language means to separate the code implementing a crosscutting concern from the functional code of a software application. Using AOP, an application consists of two parts: The *base program*, which implements the core functionalities, and the *aspects*, which implement the crosscutting concerns. Aspects are new units of modularity, which aim at modularizing crosscutting concerns in complex systems by using *join points*, *pointcuts*, and *advices*.

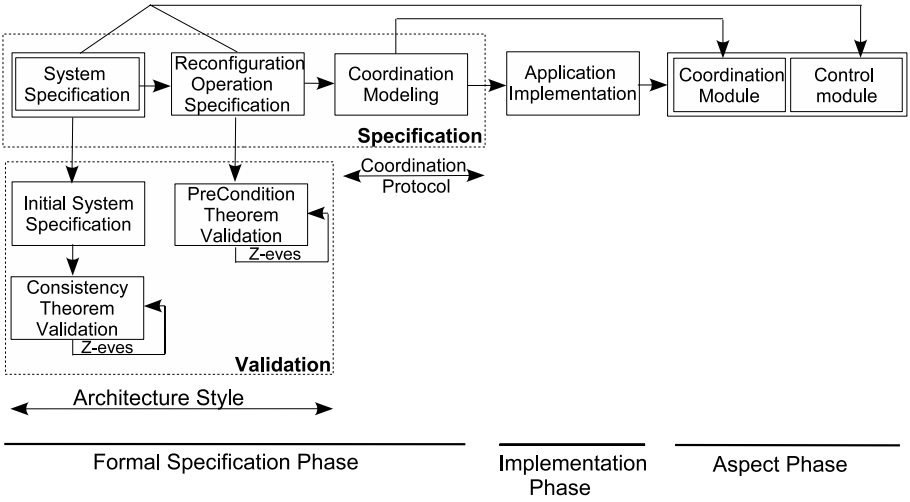


Fig. 1. Approach phases

Join points are well-defined points in the execution of a program. In AspectJ [16], which is an aspect-oriented extension to Java, join points correspond to method calls, constructor calls, field read/write, etc. The pointcut allows to select a set of join points, where some crosscutting functionality should be executed.

The advice is a piece of code implementing a crosscutting functionality, which can be associated with a pointcut. The advice is executed whenever a join point in the set identified by the pointcut is reached. It may be executed before, after, or instead of the join point at hand; this corresponds respectively to the advice types *before*, *after* and *around* in AspectJ. With an around advice, the aspect can control the execution of the original join point: it can integrate the further execution of the intercepted join point in the middle of some other code (using *proceed*).

3 Proposed Approach

We propose a centralized approach for controlling[2] the software architecture evolution of component based applications, which combines formal methods and Aspect-Oriented Programming. Our approach consists to specify, validate and enforce architectural constraints in distributed applications at runtime. We distinguish three phases: formal specification, base code implementation, and aspect code implementation (Fig. 1).

The formal specification phase consists in specifying and validating the

[2] The term control in our paper has no relation with the control theory

architectural style (i.e. definition of component and architectural constraints) using the Z notation and model the coordination protocol using Petri nets. The base code implementation phase consists in implementing the functional code of the application using any Java-based component model. The aspect code implementation phase is the verification phase which allows the connection between the formal specification and the application.

In our approach, we can define constraints in the architectural style and coordination protocol without modifying any functional code of the application. To do that, we just model a new Z specification and a new Petri net. In this way, we can provide a better control and a strong reuse of code. Using aspects in our approach allows us to separate the control and coordination code from the functional code of the application, which reduces the complexity of distributed applications, and bridges the gap between the application implementation and its formal specification. Moreover, if the formal specification changes, a few well-defined modules need to be changed in a non invasive way, namely the aspects.

3.1 Formal specification phase

Formal specification provides a very effective means for a precise and unambiguous description of a software architecture. This phase consists of three steps: The first is the formal specification of the system in terms of components, relations between them and the architectural constraints. The second step is the specification and validation of the different reconfiguration operations (i.e., adding, deleting, duplicating, connecting and disconnecting components). The third step consists in modeling the coordination protocol which describes the execution order of reconfiguration operations using Petri nets. In the two first steps, we follow the Z-based approach presented in [18], which covers both the static aspect (i.e., system specification) and dynamic aspect (i.e., specification of reconfiguration operations) of software architectures.

System Specification: The system specification consists in defining the types of components, the types of relations between components, and the architectural properties specified in terms of first-order predicates. The system structure is specified using the following Z schema. C_i, R_{ij} define respectively the component and the relation between them, $Cstr_i$ represents an architectural constraint. Each component defined in the system schema must be already specified using a Z schema and include the internal behavior bhr_i in terms of predicate logic.

Component$_i$

$att_i : Type_i, \ldots$

$bhr_1;\ \ldots;\ bhr_n$

System

$C_i : Component_i$

$C_j : \mathbb{F} \setminus \mathbb{P} \setminus \text{seq } Component_j$

$R_{ij} : Component_i \leftrightarrow Component_j$

$Cstr_1;\ ...;\ Cstr_n$

To verify that our system does not contain any contradictions between the defined constraints, we should verify the system consistency by ensuring that at least one valid state exists [32]. This verification is specified by the following consistency theorem. *SystemInit* corresponds to a Z schema which describes a valid state of the system. In order to validate and reason about the architectural style, we use the tool Z/EVES [20], which supports syntax and type checking as well as theorem proving.

SystemInit

System

$C_j = \{\ldots\}$

$R_{ij} = \{(\ldots,\ldots)\}$

Theorem *ThConsistency*

$\exists\, System \bullet SystemInit$

Specification of reconfiguration operations: This step consists in specifying the dynamic aspect of architectural styles which is defined through a set of architecture reconfiguration operations. The operations are defined as Z operation schemas and correspond to adding, deleting, duplicating, connecting, and disconnecting components. Each operation schema *Operation*$_i$ defines the pre-conditions and post-conditions of that operation. These conditions are essential to control the architectural evolution of the application. The respective operation will be executed only if its pre-conditions are evaluated to true. We specify and prove also the pre-condition theorem *PreconditionTheorem*. This theorem states the pre-conditions that must initially be satisfied to guarantee that the constraints are preserved after the execution of the operation.

Operation$_i$

$\Delta System$

$C_i? : Component_i$

---

pre_Condition

post_Condition

Theorem *PreconditionTheorem*

$\forall\, System \land C_i : Component$

$pre\_Conditions \bullet \text{pre } Operation_i$

$C_i?$ represents the input component and *pre_Condition* and *post_Condition* define respectively the pre- and post-conditions (in terms of predicate logic) of the operation *Operation*$_i$.

Coordination Protocol Modeling: The coordination protocol describes the dynamic evolution of the architecture by defining the execution

order of reconfigurations. We propose to model the coordination protocol using Place/Transition Petri nets, which allow us to prove algebraic properties such as the existence of deadlocks and livelocks. Moreover, determining whether the Petri net is live and bounded, we can prove if the evolution of software architecture finishes correctly.

We model each reconfiguration operation by a transition (i.e. already specified by Z operation schema). The enabling of a transition means that the corresponding action is conform with the coordination constraints. Consequently, the transition can be carried out and a token can be put in the next place. We use the tool *P3* [9], which allows the creation, the modeling of Petri nets, and their representation in XML.

3.2 Base code implementation phase

This phase consists in implementing the core functionality of the application without including any architecture verification code. The application can be implemented using any Java-based component model (e.g., EJB [31], CCM [23]). The structure of the application must be synchronized with the formal specification, i.e., it must comprise the components and their properties, the relations between them, as well as the implementation of the different reconfiguration operations.

3.3 Aspect code implementing phase

In order to verify and control at runtime the software architecture evolution of the component-based applications, we implement an aspect-based module. The aspect module code is separated from the functional application code in two parts: the *control module* contains aspects that check the conformity of each reconfiguration operation against the formal specification of the architectural style in Z, whereas the *coordination module* contains aspects that check and enforce the coordination protocols that are modeled using Petri nets.

Control module: This module interprets the architectural style specified in Z and extracts the architectural constraints in order to subsequently verify for each reconfiguration operation whether it can be performed or not. This module is implemented using several aspects. One aspect verifies the static constraints that are specified in the system schema, such as the properties of components. In addition, for each reconfiguration operation, there is an aspect that intercepts the execution of that operation and interprets the preconditions that are specified in the system schema and the respective operation schema.

We implemented an evaluator for Z, which evaluates the logic predicate

constraints according to the system state in terms of components and their relations. The Z specifications are saved in LaTeX form [3], which facilitates the extraction of the architectural constraints. We implemented an around advice to allow/disallow the reconfiguration operation. The advice works as follow: First, it executes the constraint evaluator with the LaTeX source as input parameter. Second, it interprets the result using the keyword *proceed*. Then, it extracts the post-condition describing the system request and updates the system state. If the reconfiguration operation does not conform to the Z specification, the aspects prohibit the execution of the operation.

Coordination module: This module enforces execution orders of the architecture operations in distributed applications. Similarly to the control module, this module checks the conformity of each reconfiguration operation against the coordination protocol that is modeled as Petri nets using the tool P3, which saves in matrix form the Petri net definition and the current marking in an XML file.

In this module, an aspect checks if a reconfiguration operation can be carried out by verifying whether the corresponding transition is enabled (applying R1 section 2.2). If that is the case, the aspect executes the reconfiguration operation, and after that updates the marking in the XML file (applying R2 section 2.2).

4 Case Study

To illustrate our approach, we implemented two applications. The first is a collaborative authoring system based on a client/server style. This application controls and manages shared documents that are located on a server. The authors connect to the server in order to edit and update these documents simultaneously. The second application is a patient monitoring system specified according to the publish/subscribe style. This application allows the nurses in a hospital to control remotely their patients and request patient data by sending a request to an event service, which manages the communication between nurses and bed monitors.

In this section, we explain how the collaborative authoring application is built according to our three-phase approach. The authors can have two roles: the *writer role* can modify, create, and delete sections of a document, whereas the *reviewer role* can correct a section and add annotations to it. Problems such as overlaps between sections that are accessed by different actors cannot occur because appropriate constraints that hinder such problems are specified formally and enforced by appropriate control aspects.

[3] Z/EVES v. 2.3 exports the specification in LaTeX file based on Z-eves.sty

4.1 Formal specification phase

System Specification: The collaborative authoring system consists of four components. The shared document *SharedDoc* is accessible to all actors, who are authorized either as Writer or as Reviewer. The shared document is defined as a sequence of sections *Section* (specified by the position of the first and last characters of it in the whole document), so that there is no overlap between sections. This constraint [C1] is specified in predicate form as shown below:

SharedDoc

$$section : \text{seq}\, Section$$

$$\forall\, i : \mathbb{N} \mid 1 \leq i < \#section \quad [C1]$$
$$\bullet\ (section(i+1)).firstCharacter$$
$$= (section(i)).lastCharacter + 1$$
$$\ldots$$

Our system specified in the schema below, consists of writers, reviewers, a shared document and relations between the authors and sections. This relation will be established only between the authors who belong to the system and the sections of the shared document. The conditions [C2,C3] are preserved by verifying the domain *dom* and the range *ran* of each relation. We restrict the number of writers [C4], and we require that a writer can connect only to one section at a given point of time [C5]. To ensure that two actors are never connected simultaneously to the same section, we specify the constraint [C6] in the schema below.

CollaborativeAuthoringSystem

$$writers : \mathbb{F}\, Writer$$
$$sharedDoc : SharedDoc$$
$$WriterSection : Writer \leftrightarrow Section$$
$$\ldots.$$

$$\text{dom}\, ReviewerSection \subseteq reviewers \quad [C2]$$
$$\text{ran}\, ReviewerSection \subseteq \{s : Section \mid s \in \text{ran}\, sharedDoc.section\} \quad [C3]$$
$$\#writers < 5 \quad [C4]$$
$$\forall\, w : writers \bullet \#(WriterSection(\!|\, \{w\} \,|\!)) \leq 1 \quad [C5]$$
$$\forall\, r : reviewers;\ w : writers;\ s : Section \mid s \in \text{ran}\, sharedDoc.section$$
$$\bullet\ (r, s) \notin ReviewerSection \lor (w, s) \notin WriterSection \quad [C6]$$
$$\ldots$$

In order to verify the consistency of the specified system, we define a valid system state *InitCASystem* and we specify and prove the theorem

ConsistencyCASystem. Our system specification is consistent and does not contain any conflict since the following theorem was proved by Z/EVES.

Theorem *ConsistencyCASystem*

$$\exists\, CollaborativeAuthoringSystem \bullet InitCASystem$$

Specification of reconfiguration operations: In the following, we specify and validate all reconfiguration operations of our system (e.g., insert writer, connect writer, disconnect writer, delete section, ...). The operation schema *InsertWriter* specifies the addition of new writer without connecting the writer to any section of the shared document. The constraint [C7] specifies that the new writer $w?$ should not be one of the writers that are already present in the system. We specify also the post-conditions [C8,C9] of the operation in terms of components and their relations.

We proved the theorem *PreInsertWriter*, which preserves the system properties while adding a new writer. This operation is conform to the system constraints described in the style schema *CollaborativeAuthoringSystem*.

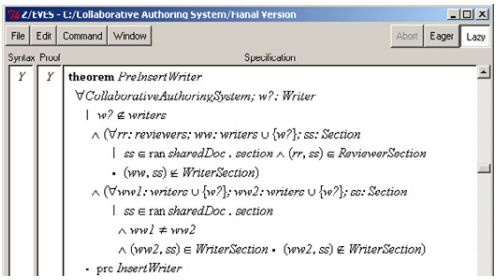


Fig. 2. Validation of the Writer insertion using Z/EVES

InsertWriter

$\Delta\, CollaborativeAuthoringSystem$

$w? : Writer$

$w? \notin writers$	[C7]
$writers' = writers \cup \{w?\}$	[C8]
$WriterSection' = WriterSection$	[C9]
...	

Modeling coordination protocols: In our collaborative authoring system, the writers can create, modify, and delete sections. Then, the reviewers can correct these sections and add annotations. To enforce the activity order specified above, we define a simple coordination protocol, which requires that each section must be created or modified by a writer before it becomes accessible to reviewers for correction. In addition, after a section is corrected, the next reviewer cannot revise it before an author modifies it. These constraints are expressed using the following Petri net.

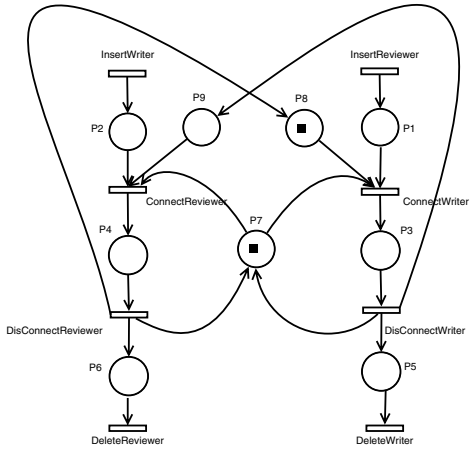


Fig. 3. Coordination protocol example

Each connection and disconnection is modeled by a Petri net transition. At execution time, actions are only executed if their corresponding transitions are possible. In the initial state, there is a token in the places $p7$ and $p8$ and consequently the transition *ConnectWriter* is enabled. Thus, a writer can connect to the section. After the disconnection of the writer, no other writer can connect because there is no token in $p8$. However, a reviewer can connect because there is a token in $p7$ and $p9$.

4.2 Base code implementation phase

Our collaborative authoring application is implemented as a Client/Server application. The functional level comprises only code providing the core functionalities such as editing, i.e., access control to the shared documents and control of the architecture evolution is out of scope. For instance, in this phase two writers can modify simultaneously the same section of a document.

4.3 Aspect code implementation phase

Control module: The control module manages the evolution of the architecture since new users can connect and/or disconnect to the documents during the execution of the application. In the architectural style of our application, we specified a static constraint expressing that each client can modify one section only at a given point of time (constraint [C5] in the system schema). Therefore, an aspect is necessary to prohibit clients from locking more than one section simultaneously.

Moreover, the architectural style specification disallows overlapping between the sections. This requirement is enforced by an appropriate aspect,

which checks if the section requested by an actor does not overlap with sections that are locked by other actors. These control operations are ensured by an aspect that interprets the architectural constraints specified in the style schema *CollaborativeAuthoringSystem*.

In addition, we have specified a dynamic constraint on the connection of a new writer. Before a new writer can connect, an aspect interprets the formal pre-conditions generated from the operation schema *connectWriter* and the style schema *CollaborativeAuthoringSystem*. This aspect defines a pointcut that selects all calls to the method *ConnectWriter*, which allows an actor to access the shared document as writer. The aspect uses an *around* advice to allow/disallow a new writer to connect.

```
1  public aspect SectionAccessControl {
2  pointcut permitted(Server S):
3  call ( public * ConnectWriter(..)) & target(S)
4      void around (Server S): permitted(S){
5          try{
6              S.parsingXMLPetriNet();
7              if (S.isTransionEnabled(ConnectWriter)) {
8                  S. resultat ="Allow";
9                   ...
10                 proceed(S);
11                 Info . updateXMLPetriNet(ConnectWriter);
12             }
13             else  {
14                 S. resultat ="Disallow";
15                  ...
16                 }
17             }
18         }catch(Exception e){}
19     }
20 }
```

Listing 1: Coordination Aspect Skeleton

Coordination module: In the collaborative authoring system, the coordination aspects enforce that actions that are executed on the shared document are according to the predefined coordination protocol (Fig. 3). Before each connection or disconnection to a section, the coordination aspects insure that a writer cannot modify a section before a reviewer corrects it and that two reviewers cannot correct the same section simultaneously.

In the following, we present a skeleton of an aspect which checks whether a section can be executed using the corresponding transition in the Petri net. The aspect *SectionAccessControl* defines a pointcut, which selects calls to the method *ConnectWriter* (lines 2 and 3 in Listing 1). The around advice of this aspect checks according to the XML representation of the Petri net if the transition *ConnectWriter* is enabled (line 7). If that is the case, the actor gets

access to the document, i.e., the operation is executed (using *proceed* as shown in line 10) and the advice saves the new marking describing the new system state. If not, the aspect prohibits the execution of the method *ConnectWriter* (lines 13-16).

5 Related Work

Several formal methods have been used for the specification of software architectures. We report in this section on the approaches that cover the static, the dynamic and the coordination aspects of software architecture.

Architecture Description Languages (ADLs) [19] provide means to describe the architecture of a software system. However, most ADLs are limited to the description of predefined dynamism [15]. They support only systems where the possible reconfigurations are at design time. In addition, with the exception of Wright, most ADLs do not have a formal basis. Many works introduce aspect-orientation to the specification level (e.g. ADL, UML [7], etc.). For instance, AO-ADL [27] is an aspect-oriented architecture description language, which models crosscutting concerns using components and provides a mechanism for defining aspect-oriented connector templates. In our current work, we are not interested in defining aspects at such a high level; we just use aspects to enforce formal architectural constraints at the implementation level.

We classify works on formal specification of software architectures according to the used techniques into three classes: based on logic, on graphs, and on process algebras. Some works used *logic*-based methods e.g., Temporal Logic [1] and Z notation [5]. The first-order logic covers only the static aspect of software architecture and pre- and post-conditions of architecture reconfiguration operations, whereas temporal logic can express at a very high level some coordination properties and temporal constrains. Other approaches use *process algebra*-based methods e.g., CSP [14] and the π-calculus [24] to model architectural dynamism with mobile processes. Other works are interested in *graphs*-based approaches to specify the static aspect of software architecture e.g., graph grammars [21] and graph Transformation [11]. These approaches do not cover the three aspects in software architecture (static, dynamic and coordination). In order to solve this problem other works propose multi-formalism approaches combining more than one formal language.

We are interested in works that combine the Z notation or Petri nets with others formal methods. ObjectZ [4] were combined with different process algebra Z/CCS [8], Z/CSP [12], and OZ/CSP [29] and with temporal logic [4]. In these approaches, Z and ObjectZ are used to specify the architectural

[4] Object-Z(OZ) is an object-oriented extension of the formal specification language Z

constraints (i.e. static aspect). However, they do not propose any formal solution to check if the architectural constraints specified in Z are preserved when the architecture evolves. In our approach, we can specify and prove the pre-condition theorem by using the Z/EVES theorem prover. Petri nets and colored Petri nets [5] were combined with different type of temporal logic [10,28] in order to define the architecture constraints and architecture evolution using temporal constraints and specify coordination constraints using Petri nets. However, Z, which is based on predicate logic and set theory allows a low-level description of architectural invariants. To support the temporal properties in our approach, we plan to combine linear temporal logic and Z notation.

In the following, we focus on approaches to the enforcement of architectural constraints. Yan et al. [33] propose an approach to discover the architecture of a system using dynamic analysis. This approach provides a tool called DiscoTect based on online monitors, which are used for system observation in order to describe inconsistencies between implementation and architecture. ArchJava [2] is an extension to Java that seamlessly unifies software architecture with implementation, ensuring that the implementation conforms to the architectural constraints. It extends a practical implementation language to incorporate architectural features and enforce communication integrity. This approach allows to enforce architecture constraints but it focuses only on communication integrity. It also does not support other types of architectural reasoning, such as reasoning about coordination protocols and architectural styles. SonarJ [13] is a commercial Eclipse plug-in [6] allowing the enforcement of architectural constraints in Java programs. The tools mentioned in this paragraph allow an efficient enforcement of architectural constraints but do not have any formal basis. In addition, crosscutting concerns are not supported by these tools.

6 Conclusion

The main contribution of this paper is the control of software architecture evolution in component-based applications in modular way using aspects. Our approach combines Aspect-Oriented Programming and formal methods and enables a reliable and modular verification. The reliability of our approach is ensured by the formal specification and validation of architectural constraints using Z and Petri nets. The use of an aspect-based control module in our approach, improves the modularity and the reusability of control code, as this code is well-modularized using aspects and separated from business logic.

[5] Colored Petri nets are a high-level extension of petri nets
[6] http://www.eclipse.org

As future work, we will study expressive pointcut languages such as [3,6,25], which allow the expression of temporal relationships in the pointcut to, e.g., express that a certain operation must be called before another. We will also investigate whether and to what extent the usage of such pointcut languages would replace the usage of Petri nets in our approach. We will also target automatic generation of control and coordination aspects.

Acknowledgments

We would like to thank Mira Mezini for several helpful discussions and comments on earlier drafts of this paper.

References

[1] Aguirre, N. and T. Maibaum, *A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems*, in: *Proc. of the 17th IEEE International Conference on Automated Software Engineering* (2002), pp. 271–274.

[2] Aldrich, J., C. Chambers and D. Notkin, *Archjava: connecting software architecture to implementation*, in: *In Proc of the 24th International Conference on Software Engineering* (2002), pp. 187–197.

[3] Allan, C., P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam and J. Tibble, *Adding trace matching with free variables to AspectJ*, in: *Proc. of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (2005), pp. 345–364.

[4] Bussow, R., R. Geisler, W. Grieskamp and M. Klar, *The μSZ Notation Version 1.0.*, Technical report, TU Berlin, Gemany (1997).

[5] de Paula, V. C., G. R. R. Justo and P. R. F. Cunha, *Specifying and Verifying Reconfigurable Software Architectures*, in: *Proc. of the 5th International Symposium on Software Engineering for Parallel and Distributed Systems* (2000), pp. 21–31.

[6] Douence, R., P. Fradetand and M. Sudholt, *Composition, Reuse and Interaction Analysis of Stateful Aspects*, in: *Proc. of the 3rd International conference on Aspect-Oriented Software Development* (2004), pp. 141–150.

[7] Fuentes, L. and P. Sanchez, *Towards executable aspect-oriented uml models*, in: *In Proc of the 11 th International Workshop on Aspect-Oriented Modeling in conjunction with Models*, 2007.

[8] Galloway, A. and B. Stoddart, *An operational semantics for ZCCS*, in: *Proc. of the International Conference of Formal Engineering Methods* (1997).

[9] Gasevic, D. and D. Devedzic, *Software Support for Teaching Petri Nets: P3*, in: *Proc. of the 3rd IEEE International Conference on Advanced Learning Technologies* (2003), pp. 300–301.

[10] He, X., H. Yu, T. Shi, J. Ding and Y. Deng, *Formally analyzing software architectural specifications using SAM*, Journal System Software **71** (2004), pp. 11–29.

[11] Heckel, R., A. Cherchago and M. Lohmann, *A Formal Approach to Service Specification and Matching based on Graph Transformation*, in: *Proc. of the 1st International Workshop on Web Services and Formal Methods in conjunction with Coordination*, Pisa, Italy, 2004, pp. 23–24.

[12] Heisel, M. and C. Shl, *Formal specification of safety-critical software with Z and real-time CSP*, in: *Proc. of the 15th International Conference on Computer Safety, Reliability and Security* (1996), pp. 31–46.

[13] Hello2Morrow, *SonarJ*, http://www.hello2morrow.de/angebot/sonarj.php (2007).

[14] Hilderink, G. H., *Graphical modelling language for specifying concurrency based on CSP*, IEE Proceedings - Software **150** (2003), pp. 108–120.

[15] Kacem, M. H., M. Jmaiel, A. H. Kacem and K. Drira, *Evaluation and Comparison of ADL Based Approaches for the Description of Dynamic of Software Architectures*, in: *Proc. of the Seventh International Conference on Enterprise Information Systems*, Miami, USA, 2005, pp. 189–195.

[16] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An Overview of AspectJ*, in: *Proc. of the 15th European Conference on Object-Oriented Programming* (2001), pp. 327–353.

[17] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-Oriented Programming*, in: *Proc. of the 11th European Conference on Object-Oriented Programming* (1997), pp. 220–242.

[18] Loulou, I., A. H. Kacem, M. Jmaiel and K. Drira, *Towards a Unified Graph-Based Framework for Dynamic Component-Based Architectures Description in Z.*, in: *ICPS*, 2004, pp. 227–234.

[19] Medvidovic, N. and R. N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, Software Engineering **26** (2000), pp. 70–93.

[20] Meisels, I. and M. Saaltink, *The Z/EVES Reference Manual (for Version 1.5)*, Reference manual, ORA Canada (1997).

[21] Métayer, D. L., *Describing software architecture styles using graph grammars*, IEEE Transactions on Software Engineering **24** (1998), pp. 521–553.

[22] Murata, T., *Petri Nets: Properties, Analysis and Applications*, Proceedings of the IEEE **77** (1989), pp. 541–580.

[23] OMG, *CORBA Component Model*, http://www.omg.org (2002).

[24] Oquendo, F., *π-method: a model-driven formal method for architecture-centric software engineering*, SIGSOFT Software Engineering Notes **31** (2006), pp. 1–13.

[25] Ostermann, K., M. Mezini and C. Bockisch, *Expressive Pointcuts for Increased Modularity*, in: *Proc. of the 19th European Conference on Object-Oriented Programming* (2005), pp. 214–240.

[26] Pellegrini, M. C. and M. Riveill, *Component Management in a Dynamic Architecture*, The Journal of Supercomputing **24** (2003), pp. 151–159.

[27] Pinto, M. and L. Fuentes, *Ao-adl: An adl for describing aspect-oriented architectures*, in: *In Proc of the Early Aspect Workshop in conjunction with AOSD*, Vancouver, British Columbia, 2007.

[28] Rodriguez-Fortiz, M. and J. Parets-Llorca, *Using predicate temporal logic and coloured Petri nets to specifying integrity restrictions in the structural evolution of temporal active systems*, in: *Proc. of the international symposium on principles of software evolution*, 2000, pp. 83–87.

[29] Smith, G., *A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems*, in: *Proc. of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, Springer-Verlag, 1997, pp. 62–81.

[30] Spivey, M., "The Z notation: a reference manual, Second Edition," Prentice Hall International Ltd., Hertfordshire, UK, 1992.

[31] SunMicrosystms, *Enterprise Java Beans*, http://java.sun.com/products/ejb/ (2001).

[32] Woodcock, J. and J. Davies, "Using Z: specification, refinement, and proof," Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[33] Yan, H., D. Garlan, B. R. Schmerl, J. Aldrich and R. Kazman, *DiscoTect: A System for Discovering Architectures from Running Systems*, in: *In Proc. of 26th International Conference on Software Engineering (ICSE)* (2004), pp. 470–479.