An Aspect-oriented Approach to the Control of Component-based Applications

Slim Kallel[1,2], Anis Charfi[1], Mira Mezini[1], and Mohamed Jmaiel[2]

[1] Software Technology Group
Darmstadt University of Technology, Germany
{kallel,charfi,mezini}@st.informatik.tu-darmstadt.de
[2] ReDCAD Laboratory
National Engineering School of Sfax, Tunisia
{slim.kallel@isecs,mohamed.jmaiel@enis}.rnu.tn

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 control 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.
To solve these problems, we propose an approach for the control of component-based distributed applications, which combines formal methods and Aspect-Oriented Programming. In this approach, we use the Z notation for describing the architectural invariants of the application and Petri nets for modeling coordination protocols. At the implementation level, the formal specification is mapped to an aspect-based control layer, which is generated automatically. Aspects intercept architecture reconfiguration events and check according to the formal specification and the coordination protocol whether the reconfiguration events can be performed.

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. 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 [15, 16, 19]. 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 control code is not well-modularized as it is tangled with the functional code of the application and scattered across the implementation of different components. Third, the code that implements the constraints may contain contradictions that did not exist in 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 control of component-based distributed applications, which combines formal methods and Aspect-Oriented Programming (AOP) [11]. This approach cover the static, the dynamic, and the behavioral aspects of the evolution of software architectures. It fosters an organization of distributed component-based software systems in three layers: the formal specification level, the functional level, and the control level. In the formal layer, 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 functional layer, the user writes the functional code of the different components in Java. This code does not contain any control logic. In the control layer, we provide a tool that automatically generates a set of 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, since the control aspects are generated automatically, mismatches between the implementation of the application and its formal specification are less probable. 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 to controlling the evolution of component-based applications. In Section 4, we explain the formal specification of software architecture of component-based applications. Section 5 describe the mapping formal specification to aspect and the aspect generation. Section 6 reports on some related work and Section 7 concludes this paper.

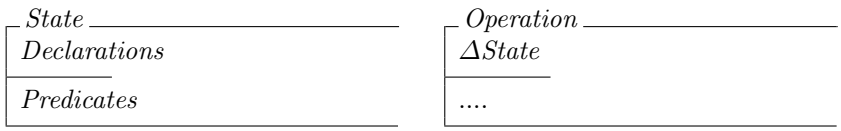
2 Background

In this paper, we make use of two well-known formal methods, namely Z and Petri nets, for specifying respectively architectural styles and coordination protocols in component-based applications. In addition, we use Aspect-Oriented Programming for modularizing the control and coordination code.

2.1 The Z specification language

The Z notation [20] 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 how this state can change. The refinement theory allows to develop a system by building an abstract model from a system design.

A specification in Z 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 should be maintained when the system is updated. This schema consists of two parts: a declaration part and a predicate part. The latter defines constraints and specifies the values of the variables that are declared in the declaration part. The operation schemes 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 *Operation* comprises the state *State* before and the state $State'$ after performing the operation. These two states are represented in the schema language by the $\Delta State$.



To validate a Z specification, we use the tool Z-EVES [12], which is an advanced analysis tool that supports syntax and type checking as well as theorem proving of Z specifications.

2.2 Petri nets

Petri nets [17] are a graphical and mathematical tool to model and analyze discrete systems. In Petri nets, the states of a system are modeled using *places* and *tokens*. The events are represented using *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 $(P \cup T) \neq \varnothing$; $F \subseteq (P \times T) \cup (P \times T)$ 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.

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

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

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

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.

To model coordination protocols with Petri nets, we use the tool *P3* [6], which supports the creation, the modeling of Petri net, and their export to XML.

2.3 Aspect-Oriented Programming

Aspect-Oriented Programming [11] is a programming paradigm, which supports the modularization of concerns that cut across the implementation of a software application, such as logging, persistence, and security.

According to the principle of *separation of concerns*, AOP provides language means to separate the code that implements 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 functionality, 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 *advice*.

Join points are well-defined points in the execution of a program. In AspectJ [10], which is an aspect-oriented extension to Java, join points correspond to e.g., 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 that implements 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 integrate the further execution of the intercepted join point in the middle of some other code using the keyword *proceed*.

3 The Approach in a Nutshell

Our approach presumes a three-phased methodology to developing distributed component-based applications. Applications that are built according to this methodology have a three-level architecture: the formal specification level, the

functional level, and the control level. In the following, the methodology and the architecture will be presented.

In the first phase of this methodology, the user specifies formally the constraints that should be satisfied when the architecture of the application evolves. In the second phase, the user provides the functional code that implements the different components. In the third phase, the user defines a mapping between the formal specification of the different reconfiguration operations and the implementation of the application. This mapping will be used by generator, which emits set of AspectJ aspects that provide the control and coordination functionalities.

In the specification phase, the Z notation is used to specify static and dynamic aspects of the software architecture, such as involved components, their types and relations, as well as invariants that must hold by any reconfiguration operations that add and/or remove components and/or relations from the system are defined using the Z notation. Coordination protocols, such as the constraints on the ordering of some operations, are defined using Petri nets. To facilitate the interpretation and the extraction of the logical predicates, the Z specifications are saved in LaTeX files using the tool Z-EVES. Moreover, the Petri nets are saved in matrix form together with the current state of the system (i.e., the current marking) in an XML file by using the tool P3.

Accordingly, the architecture of applications that are developed with the methodology described so far consists of three levels as shown in Figure 1: the formal specification level, the functional level, and the control level.

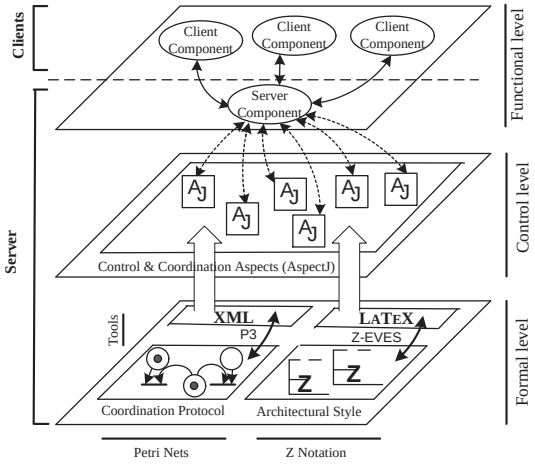


Fig. 1. Overview of the three-level architecture

To illustrate our approach, we will consider a Java application for collaborative authoring of structured documents. This application consists of shared documents that are located on a server, and clients that connect to the server to view and edit these documents. The clients can have two different roles: *writers* can modify, create, and delete sections of a document, whereas *reviewers* can correct a section and add annotations to it.

We have developed this application using the approach presented in this section. That is, problems such as overlaps between sections that are accessed by different client are avoided by specifying appropriate constraints formally (at the formal specification level) and generating AspectJ aspects (at the control level) that enforce them.

4 Formal Specification

This section presents the formal specification phase. Three kinds of specifications are produced and verified in this phase. First, the structure and the behavior of the individual components as well as the overall architecture of the system is specified and validated. Second, pre- and post-conditions for reconfiguration operations are defined and verified to ensure that the specified architectural style is maintained as the system evolves at runtime. Last but not least, valid sequences of reconfiguration operations are specified and validated.

4.1 Overall System Specification and Verification

Predicate logic is used to specify static and dynamic properties of individual components participating in the system following the component specification template shown below. In this template, att_i denotes an attribute, Spr_i and Dpr_i denote static, respectively dynamic properties of a component.

$$
\begin{array}{|l}
\hline
Component_i \\
att_1 : Type_1, att_2 : Type_2 .... att_n : Type_n \\
\hline
Spr_1, ..., Spr_n \\
Dpr_1, ..., Dpr_n \\
\hline
\end{array}
$$

The overall system specification defines a set of components, the relationships between them, and the architectural constraints that must be maintained when the system evolves. In the sample system schema shown below, c_i denotes a component instance, $Component_i$ denotes a component type, $relation_{ij}$ denotes the relation between the $Component_i$ and the $Component_j$ (as represented by the bidirectional arrow), and Apr_i denotes an architectural constraint. To verify the consistency of the system specification, it should be ensured that at least one valid initial state exists. Using Z-EVES, one can define an *InitialisationTheorem* and prove it, whereby *System* represents the system schema and *SystemInit* corresponds to a Z schema that describes the initial system state.

System

$c_i : Component_i$;...
$c_j : \mathbb{F}\ Component_j$;...
$relation_{ij} : Component_i \leftrightarrow Component_j$;...

$Apr_1, ...Apr_n$

Theorem *InitialisationTheorem*
$\exists\ System \bullet SystemInit$

In the following, we illustrate the system specification step by means of our collaborative authoring system. As an example, we consider two kinds of components in our system: shared documents and sections. A shared document is accessible to any client that is authorized either as a *Writer* or as a *Reviewer*. In the schema below, the shared document is defined as a sequence of sections that do not overlap, as specified by the predicate in the lower part of the specification of a shared document. A section is defined by the position of its first and last characters in the whole document.

Section

$firstCharacter : \mathbb{N}$
$lastCharacter : \mathbb{N}$

$lastCharacter \geq firstCharacter$

SharedDoc

$sections : \text{seq}\ Section$

$\forall\ i : \mathbb{N} \mid 1 \leq i < \#section$
$\bullet\ (section(i+1)).firstCharacter$
$= (section(i)).lastCharacter + 1$

Next, the collaborative authoring system *CollaborativeAuthoringSystem* is specified in the schema below. It consists of finite sets ($\mathbb{F}$) of writers and reviewers, a shared document and relations between authorized writers/reviewers and sections of the shared document. Conditions on the relations are preserved by verifying the domain *dom* and the range *ran* of each relation. For illustration, also the constraints $C1$ and $C2$ are given. The constraint $C1$ states that a writer or a reviewer can be connected to only one section at any point of time. The constraint $C2$ states that two actors (writers or reviewers) are never connected simultaneously to the same section. These constraints should be obeyed by any operation that changes the sets of writers or reviewers.

CollaborativeAuthoringSystem

$writers : \mathbb{F}\ Writer$
$reviewers : \mathbb{F}\ Reviewer$
$sharedDoc : SharedDocument$
$WriterSection : Writer \leftrightarrow Section$
...

$\text{dom}\ WriterSection \subseteq writers$
$\text{ran}\ WriterSection \subseteq \{s : Section \mid s \in \text{ran}\ sharedDoc.section\}$
$\forall\ w : writers \bullet \#(WriterSection(\!|\ \{w\}\ |\!)) \leq 1$ $(C1)$
$\forall\ r : reviewers;\ w : writers;\ s : Section$
$\mid s \in \text{ran}\ sharedDoc.section$ $(C2)$
$\bullet\ (r, s) \notin ReviewerSection \lor (w, s) \notin WriterSection$
.....

To verify the consistency of the *CollaborativeAuthoringSystem* schema, we define an initial system state, *InitiCASystem*, shown below. The initial state consists of two writers $w1$, and $w2$, one reviewer $r1$, and a shared document sd the latter consists of three sections $s1, s2, s3$. The proof of the consistency theorem ensures that the specification of our collaborative authoring system is consistent and does not contain any contradictions.

InitCASystem

$CollaborativeAuthoringSystem$

$writers = \{w1, w2\}$
$reviewers = \{r1\}$
$sharedDoc = sd$
$WriterSection = \{(w1, s1), (w2, s3)\}$
$ReviewerSection = \{(r1, s2)\}$

Theorem *ConsistencyCASystem*
$\exists\ CollaborativeAuthoringSystem$
$\bullet\ InitCASystem$

4.2 Specification and Verification of Reconfiguration Operations

In this step, the architectural reconfiguration operations are specified formally (examples of such operations are the insertion/removal of a component and/or a relation between components). Each reconfiguration operation is specified by means of a Z *operation schema*, which defines the input parameters ($c_i?$) as well as the pre-conditions and post-conditions. These conditions are essential to control the evolution of the architecture and to preserve certain system properties. Reconfiguration operations are executed only if their pre-conditions are satisfied. In the operation schema below, $PreCond_i$ and $PostCond_i$ denote a pre- and a post-condition of the reconfiguration operation $Operation_i$.

After specifying the reconfiguration operations formally, these operations need to be verified. To evaluate the impact of a reconfiguration operation on a constraint, we define and prove the theorem *PreCondTheorem* shown below. This theorem states the pre-conditions that must initially be satisfied to guarantee that the constraints are preserved after the execution of the operation and verify that the execution of the reconfiguration operation preserves the architectural style.

$Operation_i$

$\Delta System$
$c_i? : Component_i; \ ...$

$PreCond_i, ..., PreCond_n$
$PostCond_i, ..., PostCond_n$

Theorem *PreCondTheorem*
$\forall\ System \land c? : Component_i$
$\mid preConditions \bullet \text{pre}\ Operation_i$

Let us now illustrate the approach to specifying reconfiguration operations by means of our collaborative authoring system. We have specified and validated formally all its reconfiguration operations such as the insertion and connection of writers, reviewers, and sections. For illustration, the following schema specifies the operation *ConnectWriter*. The operation schema states that when a writer

$w?$ is connected to a section of the shared document then she should be one of the writers that are already present in the system and the section $s?$ should be already created. In order to validate the connection operation of a new writer, we use Z-EVES to prove the theorem *PreConnectWriter*, which ensures that the connection of a writer is conform to the system constraints described in the system schema *CollaborativeAuthoringSystem*. For example, this theorem states that the connection of a writer to a section requires that no reviewer is connected to that section (constraint $C2$ in page 7).

ConnectWriter

$\Delta CollaborativeAuthoringSystem$
$w? : Writer$
$s? : Section$

$w? \in writers$ $(C3)$
$s? \in sections$ $(C4)$
$WriterSection' =$
$WriterSection \cup \{(w?, s?)\}$
.....

Theorem *PreConnectWriter*

$\forall CollaborativeAuthoringSystem;$
$w? : Writer;\ s? : Section$
$\mid w? \in writers \land s? \in sections$
$\land\ (\forall r : reviewer \bullet \land\ (r, s?) \notin ReviewerSection)$
...
$\bullet$ pre *ConnectWriter*

4.3 Specification of Valid Protocols

In this step, Petri nets are used to define constraints on the execution order of reconfiguration operations that are already specified in Z. Typical behaviors in distributed component-based applications such as synchronization, mutual exclusion, conflicts, etc., can be naturally specified with Petri nets. We model each reconfiguration operation by a transition and a system state by a set of places and tokens. Enabling a transition in the Petri net means that the corresponding reconfiguration operation is conform with the constraints given the current system state. Consequently, the transition can be executed and its target places can be occupied by tokens.

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 described above, we define a 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.

In the initial state that is shown in Fig. 2, the transitions *InsertWriter* and *InsertReviewer* are always enabled. Consequently, the transition *ConnectWriter* will be enabled. Thus, a writer can connect to a section. After the enabling of the transition *DisconnectionWriter*, the writer can be deleted but she cannot connect because there is no tokens in $P8$. However, a reviewer can still connect because the transition *ConnectReviewer* is enabled.

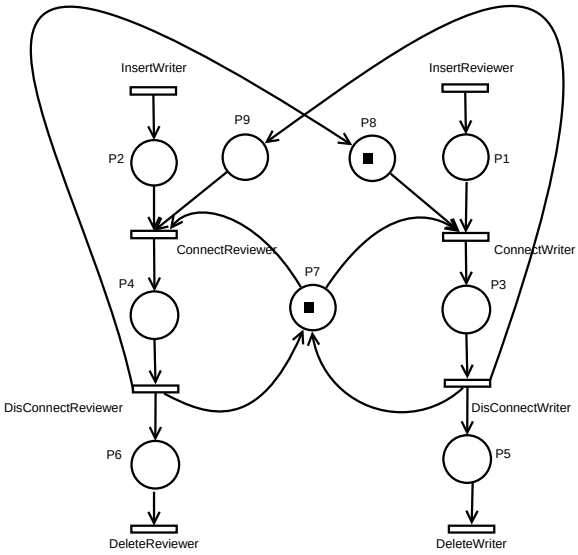


Fig. 2. A Petri net example

5 Mapping Formal Specifications to Code

This section describes how formal specifications are mapped to code. The mapping of the specifications pertaining to individual components and to the overall system architecture are done manually. The focus of this paper is on mapping cross component specifications related to reconfiguration operations and their valid protocols. This mapping is done semi-automatically by translating specifications to aspects in the AspectJ language.

5.1 Components Implementation

In this phase, the user provides the functional code that implements the different components of the application. The functional code of the components can be implemented using Java or a Java-based component model such as EJB [21]. We focus on applications that are based on a client-server architecture.

The functional code should be conform with the formal specification of the components and their relations. For example, if the formal specification states that a certain component has some attribute then the class that implements that component should have a field that matches that attribute. However, enforcing the formal specifications pertaining to individual components is out of the scope of this paper. Our focus is rather on the enforcement of cross-component control and/or coordination invariants, as elaborated in the following subsection. It is important to emphasize that in our approach the functional code does not contain any control and/or coordination functionality.

Our collaborative authoring application is implemented as a client-server application. The functional level comprises only code providing the core functionalities. The writers can create shared documents and modify or delete sections. The reviewers can correct a section by adding an annotation or by modifying the section formatting (i.e., they cannot edit the text but they can change text fonts and colors). Control code, e.g., to avoid overlaps between sections, is not encoded in these components.

5.2 Mapping Cross-Component Invariants to Aspects

In order to support a generic, reliable, and modular control of component-based applications in Java, formal specifications of reconfiguration operations and valid protocols thereof are semi-automatically mapped to AspectJ aspects in our approach. That is, the control layer consists of a set of aspects (one per reconfiguration operation); the pointcuts intercept the execution of operations from the component implementation layer that correspond to reconfiguration operations; the advice consult the current system state to check whether the reconfiguration operations can be executed; if this is the case, the advice execute the reconfiguration operations and update the system state accordingly. In other words, the aspects of the control level connect the formal level and the functional level in order to control the evolution of the architecture according to the formally specified constraints.

Each aspect contains two parts: the coordination part and the control part. The *coordination part* check and enforce the coordination protocols that are defined by Petri nets. This part verifies whether the respective transition in the Petri net is enabled based on the current system state. The *control part* corresponds to code that checks for the respective operation whether the preconditions formally specified in Z within the style schema and the operation schema are satisfied. If that is the case, the aspect executes the reconfiguration operation, and after that updates the state of the system. Otherwise, the aspect prohibits the execution of the operation.

In addition to Z and Petri net specifications, the generation process also takes a mapping of the reconfiguration operations to points during the execution of the functional code as an input.

```
1  <Mapping Name="CollaborativeAuthoringSystem">
2  <PointCut Reference="InsertWriter">
3        InsertW(Writer w):
4            call(public * insertW(..))&& target(w); </PointCut>
5  <PointCut Reference="ConnectWriter" Component="Section">
6        ConnectWriterToSection(Writer w,Section s):
7            call(public * connectW(..)) && target(w) && args(s); </PointCut>
8  ...
9  </Mapping>
```

Listing 1.1. Mapping formal specification to code

This mapping is provided by the developer using an appropriate XML file. For illustration, the extract shown in Listing 1.1 maps the reconfiguration oper-

ation *ConnectWriter*, defined in the formal specification, to calls to the method *connectW* (lines 5–7 in the Listing 1.1) from the functional level.

To automatically translate Z specifications to aspects, the specifications should be structured according to a meta-model, which ensures some properties. For instance, the system and its components must be specified in the Z schema form, the connections between the components must be specified in the form of a relation, etc. To verify that the Z specification is compliant with this meta-model, we translate the structure of the Z specification into an XML file. Moreover, we defined an XML Schema Definition (XSD) for the meta-model. We use DOM and SAX to verify whether the XML representation of the Z specification satisfies the XSD of the meta-model.

The code generator gathers all necessary information to generate the aspects. It extracts the properties of the components, the relations between components and the architectural constraints which are specified in the system schema and in the operation schemes.

For each reconfiguration operation, which corresponds to a Z operation schema and/or to a transition in the Petri net, the pre-conditions are translated to *around advice*, which coordinates and controls their execution, as schematically shown in line 5 of Listing 1.2. The generated advice is associated with a *pointcut* that is provided by the user in the XML mapping file (cf. line 2 in the Listing 1.2).

The work-flow of the generated around advice is as follows. The advice first checks whether the Petri net transition for the corresponding reconfiguration operation is enabled. For that purpose, the generated advice contains a call to the method *isTransitionEnabled* (line 8 in the Listing 1.2), which applies the rule R1 (cf. Section 2.2). Next, the advice checks whether all preconditions[1] of the corresponding reconfiguration operation are fulfilled. If one of the generated pre-conditions is not fulfilled the operation will not be executed (lines 22-26). Otherwise, the operation will be executed by calling *proceed* (line 23).

The constraints specified in the operation schema (line 11) and the constraints without quantifications (line 14) specified in the system schema are evaluated. Next, constraints that use quantification operators are evaluated by calling auxiliary methods generated for them (e.g., method $checkConstraintC_i$ in Listing 1.2).

In addition to aspects, the generator emits a Java class that stores the current state of the system in terms of components (characterized by their attributes) and the relations between them. Each generated aspect implements a method *updateSystemState* (line 33), which updates the current state of the system (including the marking by applying the rule R2 (cf. Section 2.2)), if the reconfiguration operation can be executed. This method is called in the aspect after *proceed* (line 24 in Listing 1.2).

[1] The translation of the preconditions of reconfiguration operations into Java code makes use of a Java-based package of Z that contains classes representing the elements of the Z language such as operators, mathematical objects such as sets, relations, sequences, bags, etc.

```
1    //extract the pointcut from the mapping file
2     public pointcut PointCutName (parameters1): set of join points
3
4    //The around advice for controlling the reconfiguration operation
5     void around(parameters1): PointCutName(parameters1) {
6
7        // check if the Petri net transition is enabled
8        if (isTransitionEnabled (CurentMarking, respectiveTransistion)) { ...}
9
10       // evaluate the constraints defined in the operation schema
11       if (ZOperators(parameters2, SystemState)) { ... }
12
13       // evaluate constraints without quantifications defined in the system schema
14       if (ZOperators(parameters3, SystemState)) { ... }
15
16       // call the auxiliary method for evaluating constraints with quantification
17        ...
18        if (checkConstraintCi()) { ... }
19        ...
20
21       // verify that all constraints hold and proceed if this is the case.
22        if (allConstraints) {
23            proceed(parameters1);
24            updateSystemState(parameters4);
25         }
26         else { ... }
27    }
28
29   // helper method for evaluating constraints with quantification
30   public boolean checkConstraintC_i() { ... }
31
32   // the method for updating the system state
33   void updateSystemState(parameters4) { ... }
34 }
```

Listing 1.2. Template of aspect generation

For illustration, let us consider the control and coordination aspects generated for our collaborative authoring system. Each reconfiguration operation (e.g., insert writer, connect reviewer, delete section, ...) corresponds to a Z operation schema in the formal specification of the architectural style and to a Petri net transition in a coordination protocol. For each reconfiguration operation, an AspectJ aspect is automatically generated.

The aspect shown in Listing 1.3 controls the connection of a writer to a section. This operation is specified by the Z operation schema and the Petri net transition called *ConnectWriter*. The pointcut *ConnectWriterToSection* of this aspect is taken by the aspect generator from the XML mapping file. The around advice of this aspect controls the execution of the operation *ConnectWriter*. For example, the constraints C3 and C4 from the Z operation schema *connectWriter* (page 9) are translated to calls to the method *isMemberOf* of a Z operator package that we implemented (lines 11 and 12 in the Listing 1.3). Quantified constraints such as the constraint C1, which ensures that a writer or reviewer can modify only one section at a given point of time, and the constraint C2, which disallows overlaps between sections are translated to Java code by using help methods. For instance, a help method *checkConstraintC*2 (lines 26–46) is generated to evaluate the constraint C2. This help method is called in the advice (line 15).

```
1  public aspect EnforceConstraintsForConnectWriter {
2      pointcut ConnectWriterToSection(Writer w,Section s):
3                      call(public * connectW(..)) && target(w) && args(s);
4
5      void around(Writer w, Section s): ConnectWriterToSection(w,s)
6      {
7          ...
8          if (isTransitionEnabled (CurentMarking, ConnectWriterTransistion)) { ...}
9
10         ...
11         if (isMemberOf(wInput,SystemState.writers)) { ... } //Constraint C3
12         if (isMemberOf(sInput,SystemState.sections)) { ... } //Constraint C4
13         ...
14         if (checkConstraintC1()) { ... }
15         if (checkConstraintC2()) { ... }
16         ...
17         if (allConstraints) {
18             proceed(w,s);
19             updateSystemState(w,s);
20          }
21          else { ... }
22        }
23
24     boolean checkConstraintC1() { ... }                //Constraint C1
25
26     boolean checkConstraintC2() {                      //Constraint C2
27          boolean constraints, result0, result1, result2 = true;
28          String [] Tab0 = SystemState.Newreviewers;
29          while ((Tab0.length!=0) && result0) {
30              String r = getFirstElement(Tab0);
31              String [] Tab1 = SystemState.Newwriters;
32          }
33          while ((Tab1.length!=0) && result1) {
34              String w = getFirstElement(Tab1);
35              String [] Tab2 = SystemState.Newsections;
36          }
37          while ((Tab2.length!=0)&& result2) {
38              String s = getFirstElement(Tab2);
39              constraints = Or(isNotMemberOf(w, s,SystemState.NewWriterSection),
40                               isNotMemberOf(r, s,SystemState.NewReviewerSection));
41              result2 = result2 && constraints;
42              result1 = result1 && constraints;
43              result0 = result0 && constraints;
44           }
45           return result0;
46       }
47
48       void updateSystemState(String NameW, String NameS) { ... }
49 }
```

Listing 1.3. Example of the AspectJ Code generated

The advice contains also Java code to ensure that the coordination protocols are respected. For example, we specified in the Petri net shown in Figure 2 that a writer cannot modify a section unless a reviewer has corrected it. The generated advice contains a call to the method *isTransitionEnabled* (line 8) to check wether the transition *ConnectWriter* is enabled given the current state of the system.

Moreover, the advice contains a method *updateSystemState* (lines 50), that updates the system state that is stored in a generated class, which represents the components of the systems, their relationships, and the marking of the Petri net. If all generated constraints are evaluated to true, the respective operation will be executed (using *proceed*) and the system state will be updated (lines 17–21).

6 Related Work

We report on works on formal specification of software architecture, and aspect and code generation from formal specification.

In Several works, different formal methods have been used for the specification of software architectures. Some of these works used logic-based methods, e.g., Temporal Logic [2] and Z notation [1]. Other used process algebras, e.g., CSP [8] and graphs, e.g., graph grammars [13] and graph Transformation [7]. In [13], Métayer uses a graph grammar, which is based on a mathematical model, to specify software architectures formally. This type of grammars does not support certain logical properties such as reasoning about the number of component instances and logical conditions such as absence of a communication link between two software components. In [1], Gregory et al. propose using the Z notation to analyze the architecture styles and the relation between them. However, this work focuses only on the static aspects and does not address the dynamic aspects and the evolution of the architecture.

The translation of formal specifications to code has been addressed in several works. In [18], Ramkerthik and al. discuss the generation of Java code with design contracts from an Object-Z[2] Specification. This work translates the structure of Object-Z specifications to XML and then generates a Java skeleton by processing the XML representation. However, it translates only the simple predicates and the skeletons are not executable. Jia and al. [9] propose an approach to synthesizing functional code from the UML and Z. They translate the UML models into a Z specification, which is used to generate C++ code. This work is related to ours, but it generates the preconditions code in the functional code, which poses several problems of modularity and reuse as we already explained.

In more recent works, some proposals translated formal specifications into aspect code. In [4], Bodden proposes a linear-time logic over join points to verify, during the program execution, the temporal properties of certain actions (e.g., a temporal relationship between two methods calls) by alternating a finite state whose transitions are triggered through generated aspects. Unlike our approach, the approach of Bodden focuses only on temporal dependencies and does not target the architecture of the application.

7 Conclusion

In this paper, we presented an approach, based on Aspect-Oriented Programming and formal methods, for controlling the architecture evolution of component-based distributed applications. This approach covers the static, the dynamic, and the behavioral aspects of software architecture and enables a reliable and modular control.

The reliability of our approach is ensured by formal specification and validation of architectural constraints using the Z notation and the tool Z/EVES.

[2] Object-Z is an object-oriented extension of the formal specification language Z

In addition, we use Petri nets to model the coordination protocols formally. The use of an aspect-based control layer in our approach, allows us to separate the control and coordination code from the functional code, which reduces the complexity of the application, and avoids disconnections between the application implementation and its formal specification. In addition, since the control code is well-modularized in aspect, contradictions that may be introduced by programmers by mistake, when integrating the constraints into the functional code, are less likely to happen. Moreover, with aspects, users can modify the formal specification of the application without modifying the application functional code. In addition, the approach is user-friendly as the aspects are generated automatically from the Z specifications and the Petri nets.

The automatic translation of Z specifications to Java code was quite difficult to implement because of the high complexity of the Z notation. Moreover, some constraints may use quantification operators several times. Thus, the resulting aspects would be very large. The approach of generating aspects to enforce formally specified constraints is generic, i.e., it can be applied for any component-based application. In addition, we plan to use this approach in the context of Web Service composition to ensure that interactions between a composite Web Service, its partners, and its clients respect a formally specified protocol.

Another direction for our future work is to study expressive pointcut languages such as [3, 5, 14], 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.

References

1. G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, 1995.
2. N. Aguirre 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 (ASE)*, pages 271–274, Edinburgh, Scotland, UK, September 2002. IEEE Computer Society.
3. C. Allan, 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 (OOPSLA)*, pages 345–364, New York, NY, USA, 2005. ACM Press.
4. E. Bodden. Efficient and Expressive Runtime Verification for Java. In *Proc. of the Grand finals of the ACM Student Research Competition*, San Francisco, USA, March 2005.
5. R. Douence, 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 (AOSD)*, pages 141–150, New York, NY, USA, March 2004. ACM Press.

6. D. Gasevic and D. Devedzic. Software Support for Teaching Petri Nets: P3. In *Proc. of the 3rd IEEE International Conference on Advanced Learning Technologies (ICALT)*, pages 300–301, Athens, Greece, July 2003. IEEE Computer Society.
7. R. Heckel, 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 (WS-FM)in conjunction with Coordination*, pages 23–24, Pisa, Italy, February 2004.
8. G. H. Hilderink. Graphical modelling language for specifying concurrency based on CSP. *IEE Proceedings - Software*, 150(2):108–120, 2003.
9. X. Jia and S. Skevoulis. Code Synthesis Based on Object-Oriented Design Models and Formal Specifications. In *Proc of the 22nd International Computer Software and Applications Conference (COMPSAC)*, pages 393–399, Washington, DC, USA, August 1998. IEEE Computer Society.
10. G. Kiczales, 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 (ECOOP)*, pages 327–353, London, UK, June 2001. Springer-Verlag.
11. G. Kiczales, 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 (ECOOP)*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.
12. I. Meisels and M. Saaltink. The Z/EVES Reference Manual (for Version 1.5). Reference manual, ORA Canada, 1997.
13. D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553, Juillet 1998.
14. K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In *Proc. of the 19th European Conference on Object-Oriented Programming (ECOOP)*, pages 214–240, Glasgow, UK, July 2005. Springer-Verlag.
15. P. Parnes, K. Synnes, and D. Schefstrom. A Framework for Management and Control of Distributed Applications Using Agents and IP-Multicast. In *Proc. of the IEEE Conference on Computer Communications (Infocom)*, pages 1445–1452, New York, USA, March 1999. IEEE Computer Society.
16. M. C. Pellegrini and M. Riveill. Component Management in a Dynamic Architecture. *The Journal of Supercomputing*, 24(2):151–159, 2003.
17. C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Darmstadt University of Technology, Darmstadt, Germany, 1961.
18. S. Ramkarthik and C. Zhang. Generating Java Skeletal Code with Design Contracts from Specifications in a Subset of Object Z. In *Proc. of the 5th IEEE/ACIS International Conference on Computer and Information Science(ICIS)*, pages 405–411, Honolulu, Hawaii, July 2006. IEEE Computer Society.
19. R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proc. of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 172–179, Washington, DC, USA, July 1998. IEEE Computer Society.
20. M. Spivey. *The Z notation: a reference manual, Second Edition*. Prentice Hall International Ltd., Hertfordshire, UK, 1992.
21. SunMicrosystms. Enterprise Java Beans. http://java.sun.com/products/ejb/, 2001.