Safe Integration of New Concerns in a Software Architecture

Olivier Barais*, Julia Lawall†, Anne-Françoise Le Meur* and Laurence Duchien*

*Jacquard project, INRIA/LIFL
Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq Cedex, France
{barais, lemeur, duchien}@lifl.fr

†DIKU, University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

Abstract

Software architectures must frequently evolve to cope with changing requirements, and this evolution often implies integrating new concerns. Unfortunately, existing architecture description languages provide little or no support for this kind of evolution. The software architect must modify the architecture manually, which risks introducing inconsistencies.

In previous work, we have proposed the TranSAT framework, which provides a pattern construct for describing new concerns and their integration into an existing architecture. As the interaction between the new concern and the existing architecture may be complex, it is essential that the framework ensure the coherence of the resulting architecture. In this paper, we introduce a language for specifying patterns and verifications that ensure that the concern represented by a pattern can be safely integrated into an existing architecture. The verifications comprise static verifications that check coherence properties before the architecture is modified and dynamic verifications that focus on the parts of the architecture that are affected by the pattern. As a result of these verifications, patterns can be provided as a commodity, such that a software architect can confidently apply a pattern obtained from a third-party developer.

1 Introduction

Software architecture is a key concept in the design of a complex system. An architecture models the structure and behavior of the system, including the software elements and the relationships between them. While architectures were originally specified informally [1, 7], recent years have seen the creation of a number of Architecture Description Languages (ADLs) [12], which enable an architect to create an architecture by constructing and combining increasingly complex elements. Nevertheless, the dimensions of composition and interaction provided by these languages are not sufficient to express concerns such as security that crosscut the software architecture. In these cases, integrating a new concern requires invasively modifying the ADL specification, at all points affected by the concern. These modifications are low-level, tedious and error-prone, making the integration of such concerns difficult.

To address the complexity of integrating a new concern into a software architecture, we have developed the TranSAT framework [4]. In the spirit of Aspect-Oriented Programming (AOP), TranSAT isolates the description of each concern in a separate architecture construct, the *pattern*, that is automatically integrated into an existing software architecture by a *weaver*. Analogous to an aspect, a pattern consists of the new architecture fragment to be integrated, a description of where this fragment can be applied, and a specification of the transformations that should be performed to connect this new fragment to the existing architecture. As a pattern can specify updates to multiple elements in an existing architecture, it is suitable for expressing crosscutting concerns.

In specifying the integration of a new concern into an existing architecture, the coherence of the result is a key issue. Because architectures are complex, many transformations may be needed to integrate a new concern, making the specification of the integration highly error prone. Although the global coherence of an architecture can often be checked once the architecture is complete, these verifications are expensive as they consider the entire architecture. Furthermore, the interdependencies between architecture elements may make it difficult to identify the source of any error at this point.

In this paper, we address the coherence issue by proposing a language for specifying TranSAT patterns that ensures a number of coherence properties. This language is carefully designed to make certain erroneous transformations impossible to express and allows static verification of additional coherence properties before pattern integration. There remain, however, some properties that can only be checked dynamically, when integrating a pattern into an existing architecture. For these properties, TranSAT uses in-

formation found in the pattern specification to limit the cost of the checks, by focusing on the parts of the architecture affected by the pattern, and to present error messages in terms of the pattern elements. Overall, this approach provides verification early in the architecture development process, to enable the architect to rapidly and safely integrate new concerns.

The rest of this paper is organized as follows. Section 2 gives an overview of the TranSAT framework through an example, and raises some issues related to the coherence of concern integration. Section 3 presents the proposed language for specifying patterns, and Section 4 describes the associated verifications. Finally, Section 5 describes some related work and Section 6 concludes and gives some future work.

2 Overview of TranSAT

In this section, we present an overview of the TranSAT framework, illustrated in Figure 1, through the example of a banking software architecture. We first describe the architecture and then show how to use the TranSAT framework to extend this architecture with an atomicity concern. Finally, we consider some of the issues that confront an architect when specifying a crosscutting concern.

2.1 Example

Our example banking application manages the withdrawal and deposit of money between savings and checking accounts. This application is represented by the software architecture shown in Figure 2, which is specified using the SafArchie ADL [3]. Like other ADLs, such as Darwin [10] or SOFA [16], SafArchie provides structural and behavioral descriptions of component interfaces. The structural description defines the components and the bindings between them, while the behavioral description specifies the behavioral interactions of each component with its environment.

Figure 2(a) gives the structural description of the banking architecture. The structure is described in terms of composites (`Bank`, `Clients`), components (`Manager`, `Savings`, `Checking`), ports (`p1` to `p5`), delegated ports (`dp1` to `dp3`) and bindings. Ports contain operations; for example, the operations `withdraw` and `deposit` are provided by the ports `p4` and `p5`. A port must contain at least one operation, must be part of exactly one component, and must be bound to exactly one other port, in some other component. Operations are either provided or required. Bound ports must contain compatible operations; for example, port `p2` requires the operations provided by port `p4`. Delegated ports do not contain any operations; they define the interface of a composite, exporting the operations of the composite's components.

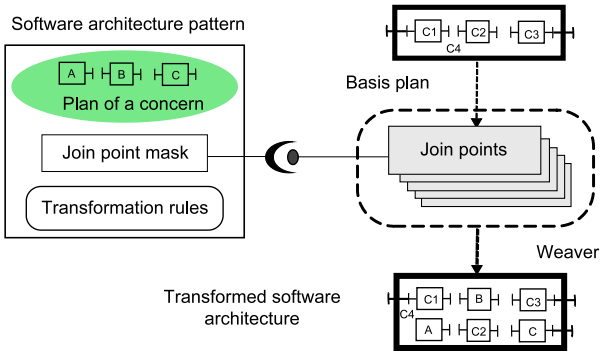


Figure 1. Overview of TranSAT

Figure 2(b) gives the behavioral description of one of the components, `Manager`. The behavior is specified in terms of an Input/Output Automaton [9] that describes the sequences of messages that a component may receive and emit. The notation used in these automata is as follows. For a provided operation `op1`, the message `?op1` represents the receipt of a request and the message `!op1$` represents the sending of the response. `?op1` must precede `!op1$`, but they can be separated by any number of messages, representing the processing of `op1`. For a required operation `op2`, the message `!op2` represents the sending of a call and the message `?op2$` represents the receipt of the response. Sending a call is a blocking operation, and thus `!op2` must always be immediately followed by `?op2$`. Using this notation, the behavior shown in Figure 2(b) specifies that when the `Manager` receives a transfer request, it makes a withdrawal from one of the two accounts and a deposit to the other one.

2.2 Integrating an atomicity concern using the TranSAT framework

The TranSAT framework manages the integration of a new concern, represented as a *software architecture pattern*, into an existing architecture, referred to as a *basis plan*. The software architecture pattern represents the new concern in terms of a *plan*, a *join point mask*, and a set of *transformation rules*. The plan describes the structure and behavior of the new concern. The join point mask defines the structural and behavioral requirements that the basis plan must satisfy so that the new concern can be integrated. The transformation rules specify the means of integrating the new plan into the basis plan. Given a software architecture pattern, the architect specifies where it should be added to the basis plan. The TranSAT *weaver* then checks that the selected point in the basis plan matches the join point mask, instantiates the transformation rules according to the architectural entities matched by the join point mask, and executes the instantiated transformation rules to integrate the new concern into

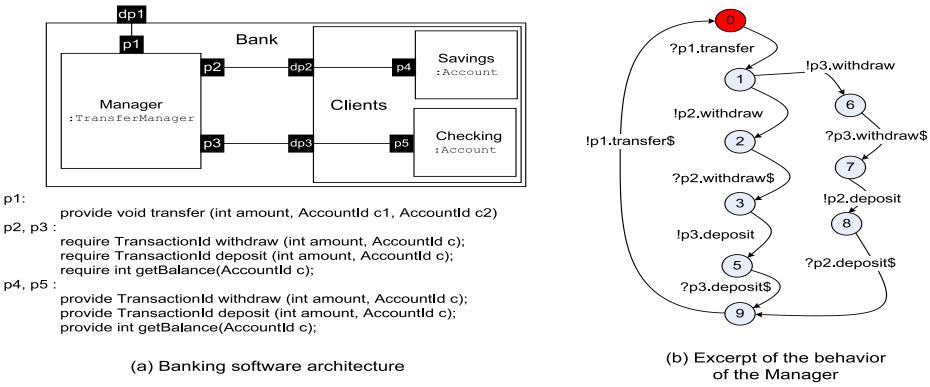


Figure 2. Banking software architecture

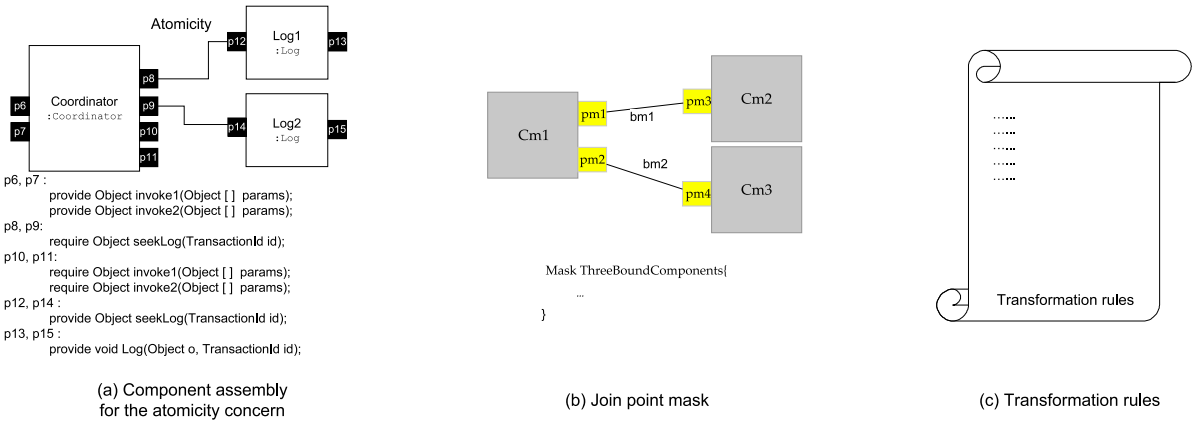


Figure 3. Architecture pattern for the atomicity concern

the basis plan. These features of TranSAT are illustrated in Figure 1.

As an example of the use of these constructs, we consider how to make the banking transactions atomic. This concern is crosscutting, in that it affects both the `Manager` and the savings and checking accounts. The architecture pattern related to atomicity is shown in Figure 3. The new plan corresponding to the atomicity concern keeps a log of certain operations and enables these operations to be rolled back when an error occurs. Specifically, the `Log` components provide operations to keep a log and to retrieve information from this log, and the `Coordinator` component triggers rollbacks when appropriate, guaranteeing the atomicity property. The join point mask specifies that this plan can be integrated in a context consisting of one component `Cm1` attached to two other components `Cm2` and `Cm3`. Some constraints (not shown) are also placed on the operations in the ports connecting these components. In the banking software architecture, the join point mask is compatible with the integration site consisting of the `Manager`, `Savings` and `Checking` components. Finally, the transformation rules connect the ports of the plan to the ports of the selected integration site, and make other appropriate adjustments. In the case of the banking architecture, the result of the integration is shown in Figure 4.

The plan, join point mask, and transformation rules shown in Figure 3 are expressed using the language that we propose for the TranSAT framework, and that is described in detail in the rest of this paper. In Figure 3, some parts of these specifications are represented as diagrams, for conciseness.

2.3 Issues

To specify the integration of a crosscutting concern, the architect must describe how to modify the component structure, behavior, and interfaces. This task is highly error prone, as many modifications are typically required, and these modifications can have both a local impact on the modified elements and a global impact on the consistency of the architecture.

Typically, a component model places a number of requirements on local properties of the individual architectural elements. For example, in SafArchie, it is an error to

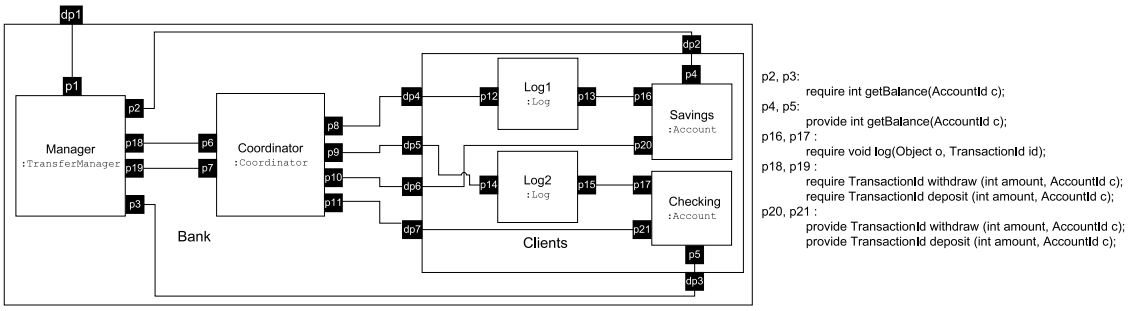


Figure 4. Transformed banking software architecture

break a binding and then leave the affected port unattached, or to remove the last operation from a port, and then leave the port empty. The construction of the behavior automaton associated with each component is particularly error prone, because it must be kept coherent with the other elements of the component and because of the complexity of the automaton structure. For example, in SafArchie, all of the operations associated with the ports of a component must appear somewhere in the component's behavior automaton. When the ADL separates the structural and behavioral descriptions, it is easy to overlook one when adding or removing operations from the other. An automaton must also describe a meaningful behavior; at a minimum that for each operation, a call precedes a return and every call is eventually followed by a return from the given operation.

The architecture must also be globally coherent. The most difficult aspect of this coherence is in the behavior of the architecture. So that the application can run without deadlock, it must be possible to synchronize the behavior of each component with that of all of the components to which it is bound by its ports. Any change in the behavior of a single component can impact the way it is synchronized with its neighbors, which in turn can affect the ability to synchronize their behaviors with those of other components in the architecture. The interdependencies between behaviors can make the source of any error difficult to determine.

3 The TranSAT's Transformation Language

In this section we present the TranSAT's transformation language for specifying the elements of a pattern: plan, join point mask and transformation rules. The component assembly shown in Figure 3 (a) is an example of a plan, showing only structural information. We present the join point mask and the transformation rules in the rest of this section. The use of the language is illustrated through the definition of the atomicity concern pattern.

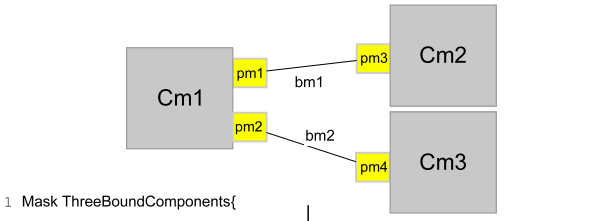
3.1 The join point mask

The join point mask describes structural and behavioral preconditions that a basis plan must satisfy to allow the integration of the new concern. It consists of a series of declarations specifying requirements on the structure and behavior of the components available at the integration site.

Figure 5 illustrates a join point mask suitable for use with the atomicity plan (Figure 3 (a)). For readability, some of the declarations are elided or represented by the diagram at the top of the figure. The diagram specifies that some component `Cm1` must be connected to two other components `Cm2` and `Cm3`. The remaining declarations define a series of placeholders for operations (line 3), specify whether these operations must be declared as provided or as required (lines 4-11) and specify that they must be associated with the ports `pm1` to `pm4` (lines 12-15). Finally, lines 16-19 ensure that the operation `opm1` is the inverse of operation `opm5` in the bound port, and similarly for `opm2` and `opm6`, `opm3` and `opm7`, and `opm4` and `opm8`. Operations are inverse if they have the opposite polarity, the same name and compatible types. In the banking architecture, these constraints would, for example, allow the architect to select the required operation `withdraw` in port `p2` as `opm1` and the provided operation `withdraw` in port `p4` as `opm5`. In this example, the join point mask does not specify any behavioral requirements. If needed, the constraints on the behavior of a component mask can be specified in terms of a sequence of messages.

3.2 The transformation rules

The transformation rules describe precisely how to integrate the new plan into a basis plan. They specify the various transformations to perform on the elements defined in the new plan and the join point mask, as well as their application order. The language provides two kinds of transformation primitives: *computation transformation primitives* and *interaction transformation primitives*. The computation transformation primitives specify the introduction of new




```
1  Mask ThreeBoundComponents{
2        … //Component and port
           declarations corres-
          -ponding to the diagram
3        OperationMask opm1, opm2,
           opm3, opm4, opm5, opm6,
           opm7, opm8;
4        opm1.polarity = require;
5        opm2.polarity = require;
6        opm3.polarity = require;
7        opm4.polarity = require;
8        opm5.polarity = provide;
9        opm6.polarity = provide;
10       opm7.polarity = provide;
11       opm8.polarity = provide;
12   pm1.operations = {opm1, opm2, *};
13   pm2.operations = {opm3, opm4, *};
14   pm3.operations = {opm5, opm6, *};
15   pm4.operations = {opm7, opm8, *};
16   opm1 = inverse(opm5);
17   opm2 = inverse(opm6);
18   opm3 = inverse(opm7);
19   opm4 = inverse(opm8);
20 … //Binding declarations}
```

Figure 5. Join point mask definition

ports and operations in primitive components, in order to adapt the component behavior. The interaction transformation primitives manage the insertion and deletion of component bindings and manage the composite content, in order to reconfigure the software architecture. Overall TranSAT is targeted towards introducing new concerns into existing architectures rather than removing existing functionalities. Thus, the language has been designed to prevent transformations that remove existing behaviors.

Computation transformation primitives

Table 1 shows the primitives used to manage the structural transformation of primitive component interfaces. These primitives allow the architect to create new ports and operations, to destroy empty ports and to move an operation from one port to another. We now present these primitives in more detail.

	Port	Operation
create	Port *Pr* in *Cp*;	Operation *Or* = *op* in *Pr*; Operation Or_1 = *op* replaces Or_2;
destroy	*Pr*.destroy();	N/A
move	N/A	*Or*.move(*Pr*);

Cp: ComponentRef, Pr: PortRef, Or: OperationRef,
op ::= *Or* | inverse(*Or*), N/A: Not applicable

Table 1. Computation transformations

Port modifications The primitive `Port Pr in Cp` creates a new port `Pr` and attaches it to the component `Cp`. The primitive `Pr.destroy()` destroys port Pr, which must be a port matched by the join point mask. As illustrated by the use of * in lines 12-15 of Figure 5, there may be incomplete information about the contents of such a port. The `destroy` primitive only actually destroys the port if it contains no operations. In this case, any binding associated with the port is destroyed as well. Note that, there is no means of moving a port to another component. Indeed, since a port may contain some operations, moving the port to another component would imply removing part of the behavior of the original component, which is not the intent of the TranSAT framework.

Operation modifications Two primitives enable the architect to create an operation: `Operation Or = op in Pr` adds a new copy of the operation `op` to the port `Pr` and `Operation` $\texttt{Or}_1$ `= op replaces` $\texttt{Or}_2$ replaces the operation $\texttt{Or}_2$ by a new copy of the operation `op` and names it $\texttt{Or}_1$. In the latter case, the new and old operations `op` and $\texttt{Or}_2$ must have the same polarity. The language also allows the architect to move an operation from one port to another if these ports are on the same component. No primitive is provided to destroy an operation or to move it to another component, to guarantee that no part of a component behavior can be removed.

Adding an operation to a port has an impact on the behavior of the associated component. When a new copy of an operation is added to a port using the operation `Operation Or = op in Pr`, the architect must explicitly specify how the messages associated with the newly added operation `op` fit into the behavior of the component to which the operation is attached. The transformation of the behavior automaton is specified using the pattern-matching syntax *template* `=>` *result*. Such a rule inserts the messages associated with the new operation, `op`, before, after, or around the calling or responding messages associated with some existing operation, `m`. The template specifies the sequence of messages on `m`, possibly separated by any sequence of messages, `x`. The result describes how messages associated with the new operation, `op`, are interleaved with this sequence. On the contrary, when one operation replaces another, the behavior update is implicit. The new operation simply inherits the old one's role in the behavior of the associated component. When an operation is moved from one port of a component to another, there is no effect on the actual behavior, although the name of the operation in the automaton, as illustrated in Figure 2 (b), is implicitly updated to reflect its new port.

The following lines illustrate the use of the automaton transformation rules:

```
?m → x → !m$ ⇒ ?m → !op → ?op$ → x → !m$;          1
?m → x → !m$ ⇒ ?m → (!op → ?op$ → x | x) → !m$;    2
```

In line 1, the template describes the receipt of a call to `m` followed by any number of messages, followed by the sending

of m's response. The result specifies that following the receipt of the call to m, the component sends a call to op and waits for the response before performing any further computation. The use of the new operation op at runtime can also be conditional. In line 2, the transformed component either calls op, waits for the response, and then performs the sequence x, or performs x alone, ignoring the added op operation.

Interaction transformation primitives

The interaction transformation primitives manage the reconfiguration of the software architecture. As shown in Table 2, operators are provided to create and destroy bindings, to create composites either at the top level or within another composite, and to move one composite Cr_1 or one component *Cp* into another composite Cr_2. There is no primitive to create a component, since new components should be specified in the plan. The destruction of a component or a composite is also not allowed in order to prevent behavior information from being removed. Finally, as a binding does not itself contain any information, there is no need for an operation to move bindings.

	Binding	Composite	Component
create	Binding *Br* = $\{Pr_1, Pr_2\}$;	Composite *Cr*; Composite Cr_1 in Cr_2;	N/A
destroy	*Br*.destroy();	N/A	N/A
move	N/A	Cr_1.move(Cr_2);	*Cp*.move(Cr_2);

Cp: ComponentRef, Cr: CompositeRef, Pr: PortRef, Br: BindingRef, N/A: Not applicable

Table 2. Interaction transformations

Example

We use the atomicity example to illustrate the use of the computation and interaction transformation primitives. In this example, integrating the new plan requires (i) interposing the Coordinator component between the original component Cm1 (instantiated as Manager in the banking case) and the operations that are to be made atomic, and (ii) inserting the Log components in front of the components Cm1 and Cm2 providing these operations (instantiated as Savings and Checking in the banking case). Figure 6 shows the rules that carry out these transformations.

In the join point mask, the operations to be made atomic are specified to be in a port that may contain other operations, *e.g.*, port pm1 contains the operations opm1, opm2, and some unknown list of operations * (line 12 in Figure 5). So that the atomicity concern does not have to take into account these other operations, lines 2-11 in Figure 6 move the operations to become atomic into newly created ports, p18

```
// Cm1 transformation                                                    1
Port p18 in Cm1;                                                         2
opm1.move(p18);                                                          3
opm2.move(p18);                                                          4
... Similarly for the port p19 and the operation masks opm3 and opm4 of pm2   5
                                                                         6
// Cm2 transformation                                                    7
Port p20 in Cm2;                                                         8
opm5.move(p20);                                                          9
opm6.move(p20);                                                          10
... Similarly for the port p21 in Cm3 and the operation masks opm7 and opm8 of pm4   11
                                                                         12
// Port destruction                                                      13
pm1.destroy();                                                           14
pm3.destroy();                                                           15
... Similarly for the ports pm2 and pm4                                  16
                                                                         17
// Coordinator transformation                                            18
Operation o6a = inverse(opm1) replaces p6.invoke1;                       19
Operation o6b = inverse(opm2) replaces p6.invoke2;                       20
... Similarly for the operations of the port p7                          21
                                                                         22
Operation o10a= inverse(opm5) replaces p10.invoke1;                      23
Operation o10b= inverse(opm6) replaces p10.invoke2;                      24
... Similarly for the operations of the port p11                         25
                                                                         26
// Introduction of p16 within Cm2                                        27
Port p16 in Cm2;                                                         28
Operation o16 = inverse(p13.log) in p16;                                 29
?opm5 → x → !opm5$                                                       30
        ⇒ ?opm5 → x → !o16 → ?o16$ → !opm5$;                             31
?opm6 → x → !opm6$                                                       32
        ⇒ ?opm6 → x → !o16 → ?o16$ → !opm6$;                             33
                                                                         34
// Introduction of p17 within Cm3                                        35
... Similarly to Cm2 for the transformation of the port p17              36
                                                                         37
// Component introduction                                                38
Coordinator.move(Cm1.parent);                                            39
Log1.move(Cm2.parent);                                                   40
Log2.move(Cm3.parent);                                                   41
                                                                         42
// Binding creation                                                      43
Binding b6 = {p18, p6};                                                  44
Binding b7 = {p19, p7};                                                  45
                                                                         46
Binding b10 = {p10, p20};                                                47
Binding b11 = {p11, p21};                                                48
                                                                         49
Binding b13 = {p13, p16};                                                50
Binding b15 = {p15, p17};                                                51
```

Figure 6. Transformation rules for the atomicity concern

to p21. This transformation may cause the ports matched by the join point mask to become empty. Accordingly, lines 13-16 apply the destroy operation to these ports, causing them to be destroyed if they are empty. When the atomicity concern is integrated into the banking software architecture, the ports matched by pm1 to pm4 are not destroyed because they contain the operation getBalance.

The ports of the Coordinator are then updated with references to the operations to be made atomic. For each port, p6, p7, p10, and p11, the generic operations invoke1 and invoke2 are replaced by the inverses of the corresponding operations in the ports p18 to

p21 (lines 18-25). These transformations implicitly update the Coordinator's behavior automaton by replacing the messages associated with the invoke operations by the messages associated with the new operations.

To insert the Log components in front of Cm2 and Cm3, new ports must be added to Cm2 and Cm3 and these ports must be instantiated with references to the log operation. We focus on the transformation of Cm2, as the transformation of Cm3 is similar. Lines 28-29 add the port p16 and copy the require counterpart of the Log component's log operation into this port. Because log is a new operation for Cm2, we must specify where it fits into Cm2's behavior. Lines 30-33 specify that Cm2 sends a call to this new operation whenever it is about to return from either of the operations to be made atomic.

The remaining rules transform the interaction between components. Lines 39-41 add the components of the plan to the basis plan. In these rules, for any outermost component or composite reference C in the join point mask, C.parent represents the parent of the element to which C is matched in the basis plan. As the component model is arborescent, each component or composite has at most one parent. If there is no parent, the enclosing transformation is not performed. Finally, lines 43-50 connect the components at the various ports. TranSAT automatically adds delegated ports, *e.g.*, dp4 in Figure 4, as needed. Applying these transformation rules to the join point between the Manager, Savings and Checking components shown in Figure 2 (a) produces the software architecture shown in Figure 4 (structural information only).

4 Safe Architecture Transformation

A goal of TranSAT is to ensure that the integration of a new concern produces a valid software architecture. Accordingly, TranSAT statically checks various properties of the pattern at creation time and dynamically checks that the pattern is compatible with the insertion context when one is designated by the architect.

4.1 Static properties and checks

Given a pattern, TranSAT first checks that its various elements are syntactically and type correct. For example, a join point mask must declare that a port contains elements of type Operation and a Binding transformation must connect two ports. TranSAT then performs specific verifications for the plan, the join point mask, and the transformation rules.

Plan TranSAT requires that the plan be a valid software architecture according to the component meta-model of SafArchie, except that it may contain unattached ports. For example, TranSAT checks that all bindings connect ports that contain compatible operations and that the automata describing the behaviors of the various components in the plan can be synchronized.

Join point mask The variables declared by the join point mask represent the fragments of the basis architecture that can be manipulated by the transformation rules. Unlike the plan, the join point mask need not be a complete architecture specification and thus TranSAT does not check that *e.g.* operations are specified for all ports or automata are synchronizable. These properties are, however, assumed to be satisfied by the elements matched in the basis architecture. TranSAT does verify the consistency of the information that is given, for example that any automata provided use operations in a manner consistent with their polarity.

Transformation rules TranSAT ensures the safety of the transformation process by a combination of constraints on the transformation language and verifications performed statically on the transformation rules.

Several features of the transformation language have been designed to prevent the architect from expressing unsafe transformations. For example, the SafArchie component meta-model requires the insertion of delegated ports whenever a binding crosses a composite boundary. TranSAT introduces these delegated ports automatically, relieving the architect of the burden of identifying the composites between two ports, reducing the size of the transformation specification, and eliminating the need to fully specify composite nesting in the join point mask. The SafArchie component model also requires that each architectural element have a parent, except for the outermost components or composites. The transformation language enforces this constraint by combining the creation of a new element with a specification of where this element fits into the architecture; for example, Port Pr in Cr both creates a new port Pr and attaches this port to the composite Cr. Finally, a common transformation is to replace an operation in a port by another operation, which requires updating both the port structure and the automaton of the associated component. The transformation language combines both operations in the declaration Operation Or_1 = op replaces Or_2.

Other safety properties are not built into the syntax of the transformation language, but are checked by analysis of the transformation rules. We consider the property that every architectural element, except the operations, must have some subelements. For example, a component must contain some ports and a port must contain some operations. This property is checked by a *containment analysis*, that also checks the usefulness of each transformation rule; for example, it is not allowed to move an element just to move it again. The analysis simulates the execution of the transformation rules on the various elements identified by the plan

Creating a port:

$$\rho[Cp \mapsto (\pi, \zeta)], J, P \vdash \texttt{Port } Pr \texttt{ in } Cp \\ \rightarrow \rho[Pr \mapsto (Cp, \emptyset), Cp \mapsto (\pi, \zeta \cup \{Pr\})], J, P \quad (1)$$

Adding an operation to a port:

$$\frac{Or_2 \notin \zeta_2}{\rho[Pr \mapsto (\pi_2, \zeta_2)], J, P \vdash \texttt{Operation } Or_1 \texttt{ = } Or_2 \texttt{ in } Pr \\ \rightarrow \rho[Or_1 \mapsto (Pr, \emptyset), Pr \mapsto (\pi_2, \zeta_2 \cup \{Or_1\})], J, P} \quad (2)$$

$$\frac{Or_2 \notin \zeta_2}{\rho[Pr \mapsto (\pi_2, \zeta_2)], J, P \vdash \texttt{Operation } Or_1 \texttt{ = inverse(}Or_2\texttt{) in } Pr \\ \rightarrow \rho[Or_1 \mapsto (Pr, \emptyset), Pr \mapsto (\pi_2, \zeta_2 \cup \{Or_1\})], J, P} \quad (3)$$

Moving an operation between ports:

$$\frac{Or \in J \qquad \pi_2 = \pi_3}{\rho[Or \mapsto (\pi_1, \zeta_1), Pr \mapsto (\pi_2, \zeta_2), \pi_1 \mapsto (\pi_3, \zeta_3)], J, P \vdash Or\texttt{.move(}Pr\texttt{)} \\ \rightarrow \rho[Or \mapsto (Pr, \zeta_1), Pr \mapsto (\pi_2, \zeta_2 \cup \{Or\}), \pi_1 \mapsto (\pi_3, \zeta_3 - \{Or\})], J - \{Or\}, P} \quad (4)$$

Figure 7. Excerpt of the containment analysis

and the join point mask. At the end of the analysis, every element must have at least one subelement except operations and join point mask elements for which no subelements are initially specified.

The containment analysis is defined in terms of judgments of the form $\rho, J, P \vdash s \rightarrow \rho', J', P'$. The environment ρ maps each element to a tuple (π, ζ) indicating its parent π and its children ζ. The lists J and P indicate the elements in the join point mask and the plan, respectively, that have not yet been moved. To ensure usefulness, `move` and `destroy` are only permitted on these elements.[1] The term being analyzed is represented as s. The results of the analysis are a new environment ρ' and lists J' and P'. Rules are expressed as inferences, as shown in Figure 7, with hypotheses above a horizontal bar and the judgement that follows from these hypotheses below the horizontal bar. The horizontal bar can be omitted if there are no hypotheses. The notation $\rho[x \mapsto (\pi, \zeta), ...]$ represents the extension of the environment ρ by the environment $[x \mapsto (\pi, \zeta), ...]$. The analysis rules are applied successively to each of the transformation rules, starting at the first one with values of ρ, J, and P that correspond to the information found in the plan and the join point mask. Each subsequent transformation rule is analyzed with respect to the ρ', J', and P' produced by the analysis of the previous one.

Figure 7 shows the rules associated with ports. The rule for creating a port (rule (1)) updates ρ with an entry for the new port and updates its parent component with the new port as an additional child. The rules for adding an operation to a port (rules (2) and (3)) are similar, but additionally check that the operation or its inverse is not already part of the destination port. Finally, the rule for moving an operation from one port to another (rule (4)) checks that the operation is part of the join point mask and that the parent component of the new port is the same as the parent component of the current port. In the result, the operation is removed from the list of elements in the join point mask to prevent the operation from being moved yet again. A transformation is accepted by the analysis if in the final environment, for every element other than an operation or an element in the join point mask starting with an empty list of children, the list of children ζ is nonempty.

A similar analysis checks various properties of bindings: every port is connected to some other port by a binding, the connected ports are not part of the same component, the operations of the connected ports are compatible, etc. Another analysis checks that for each component, the automaton and the set of operations in the various ports are kept consistent.

4.2 Dynamic checks

An architect integrates a pattern by designating a fragment of the existing architecture to which the pattern should be applied. TranSAT checks that the fragment matches the join point mask, to ensure that the fragment satisfies the assumptions under which the safety of the transformation rules has been verified. However, because the join point mask does not describe the entire basis architecture, the static checks of the different elements of the pattern are not sufficient to guarantee the correct integration of a new plan into a basis plan. Consequently, dynamic verifications of some structural and behavioral properties of the architecture are performed during the integration process.

The dynamic structural verification consists of checking the compatibility between the newly connected ports, according to the definition of the port compatibility of SafArchie. Concretely, based on transformation rules that have

[1] `move` and `destroy` are only allowed on elements named in the join point mask, *i.e.*, those in the list J, as these elements represent the existing architecture and are thus not under the control of the developer of the pattern. Composites and components can also be moved out of the plan, *e.g.*, to be placed within a composite created by the transformation rules.

been applied, the analysis builds a list containing the newly created connections as well as the connections between ports that have been modified by the transformations. For each of these connections, the connected ports are verified to contain compatible operations. The other connections do not need to be checked as they are not affected by the transformations and their correctness has been previously verified during the analysis of the basis plan or the pattern plan.

Adding new components and behaviors to a fragment of an architecture can change the synchronization at the interface of the fragment, and thus have an effect on the synchronization of the rest of the architecture. The use of a software architecture pattern localizes the modifications to a specified fragment of the existing architecture. The process of resynchronization thus starts from the affected fragment and works outward until reaching a composite for which the interface is structurally unchanged and the new automaton is bisimilar to the one computed before the transformation. The bismilarity relation ensures that the transformation has no impact on the observable behavior of the composite, and thus the resynchronization process can safely stop [15].

If the transformation of the architecture fails, any changes that were made must be rejected. Before performing any transformations, TranSAT records enough information to allow it to roll back to the untransformed version in this case.

4.3 Assessment

In Section 1, we observed that the architect who integrates a new concern with the original version of TranSAT, without the new transformation language, can use the tools of SafArchie to check the validity of the resulting architecture after the integration is complete. This approach, however, can give imprecise error messages, because the resulting architecture does not reflect the transformation step that caused the problem, and can be time consuming, due to the automaton synchronization that is part of this validation process. In this section, we briefly describe how TranSAT addresses these issues.

Because the static verifications have a global view of the transformations that will take place, they can pinpoint the transformation rules that can lead to an erroneous situation. For example, if an operation is moved from a port of the join point mask, the port may become empty, resulting in an erroneous software architecture. While SafArchie would simply detect the empty port, TranSAT can, via an analysis of the complete set of transformation rules, detect that there is a risk that a port contains only one operation, that a `move` is performed on the operation in this port, and that a `destroy` is not subsequently applied to this port. Using this information, TranSAT can inform the architect of problems in the transformation rules, before any actual modification of the architecture has taken place. Obtaining this feedback early in the integration process can reduce the overall time required to correctly integrate the new concern.

Because the dynamic verifications are aware of the exact set of components that are modified by the integration, they can target the resynchronization of automata accordingly. As synchronization is expensive, reducing the amount of resynchronization required can reduce the amount of time required to integrate a new concern, making it easier for the architect to experiment with new variants.

5 Related Work

Previous work in the software architecture domain has shown the interest of using ADLs to formally establish the consistency of a software architecture [1, 7, 12]. For example, the ADLs Wright [2] and Darwin [10] support the specification and analysis of component communication protocols. To analyze a component assembly, Wright defines a set of standard consistency and completeness verifications that are defined precisely in terms of an underlying CSP (Communicating Sequential Processes) model, and can be checked using standard model checking technology. Darwin models dynamic distributed systems and uses the FSP (Finite State Process) language to specify system behavior [11]. Both ADLs illustrate the interest of specifying the structure and behavior of components and connectors. However, such ADLs are commonly conceived around the dimensions of composition and interaction. Consequently, concerns that crosscut several architectural elements cannot be easily specified and incrementally integrated.

The aspect paradigm provides another way to specify a composition semantics between software modules. Originally targeted to the implementation phase of the software life-cycle, there has been recently some investigation of the use of aspects at the design stage. Called *Early Aspects* or Aspect Oriented Modeling [14], this work focuses on modularizing crosscutting design properties. A global model results from merging (or "weaving") these aspects. For example, the Composition Pattern [5] combines a collection of subject-oriented models for composing separate and overlapping concerns in UML (Unified Modeling Language) diagrams. Other approaches promote the use of a transformation language inspired by Model Driven Architecture [13] to specify the weaving. For example, the C-SAW weaver framework provides a generalized transformation engine for manipulating models [8]. Like these approaches, TranSAT proposes to define a global software architecture from a set of plans, whose composition is specified using a dedicated transformation language. However, contrary to the other approaches, the language used by TranSAT is designed to be able to guarantee the consistency of the transformed software architecture.

6 Conclusion

The TranSAT approach offers a solution to relieve the architect of manually integrating a new concern into a software architecture. This solution consists of factorizing the implementation of the new concern into a separate unit, called a pattern, that can be independently understood and automatically integrated at multiple integration sites. To improve the safety of this approach we have designed a pattern language that prevents some erroneous concern integrations from being expressed and detects others by static and dynamic verifications. The resulting safety guarantees mean that an architect can use a pattern provided by a third party developer with confidence that the concern will be integrated correctly.

These tools and verifications have been implemented to form a complete software architecture development environment. Future work includes the investigation of extending the framework to target other ADLs, such as AADL [6], Wright and Darwin. This extension will require studying the verifications that can be performed on these other ADLs during the transformation process.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 334–367, London, UK, 2002. Springer-Verlag.

[2] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[3] O. Barais and L. Duchien. SafArchie studio: An ArgoUML extension to build safe architectures. In Pierre Dissaux, Mamoun Filali Amine, and Pierre Michel, editors, *Architecture Description Languages*, pages 85–100. Springer, 2005.

[4] O. Barais, L. Duchien, and A.-F. Le Meur. A framework to specify incremental software architecture transformations. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 62–70. IEEE Computer Society, September 2005.

[5] S. Clarke and R. J. Walker. Composition patterns: an approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 5–14. IEEE Computer Society, 2001.

[6] P. H. Feiler, B. Lewis, S. Vestal, and E. Colbert. An overview of the SAE architecture analysis & design language (AADL) standard: a basis for model-based architecture-driven embedded systems engineering. http://www.laas.fr/FERIA/SVF/WADL04/.

[7] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[8] J. Gray, J. Zhang, S. Roychoudhury, and I. D. Baxter. C-saw and genaweave: a two-level aspect weaving toolsuite. In J. M. Vlissides and D. C. Schmidt, editors, *OOPSLA Companion*, pages 27–28. ACM, 2004.

[9] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[10] J. Magee. Behavioral analysis of software architectures using ltsa. In *Proceedings of the 21st international conference on Software engineering*, pages 634–637. IEEE Computer Society Press, 1999.

[11] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 35–50, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[12] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[13] Object Management Group (OMG). MDA (Model Driven Architecture). http://www.omg.org/mda/.

[14] A. Rashid, A. M. D. Moreira, and B. Tekinerdogan. Special issue on early aspects: aspect-oriented requirements engineering and architecture design. *IEEE Proceedings - Software*, 151(4):153–156, 2004.

[15] R.J. van Glabbeek. The linear time - branching time spectrum I. The semantics of concrete, sequential processes. In A. Ponse & S.A. Smolka J.A. Bergstra, editor, *Handbook of Process Algebra*, pages 3–99. Elsevier, 2001.

[16] S. Visnovsky. *Modeling Software Components Using Behavior Protocols*. Phd. thesis, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, December 2002.