A Graph Transformation Approach to Introducing Aspects into Software Architectures

Md Nour Hossain
Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, Ontario, Canada
hossaimn@mcmaster.ca

Wolfram Kahl
Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, Ontario, Canada
kahl@mcmaster.ca

Tom Maibaum
Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, Ontario, Canada
tom@maibaum.org

ABSTRACT

While aspect-oriented programming (AOP) addresses introduction of "aspects" at the code level, we argue that addressing this at the level of software architecture is conceptually more adequate, since many aspects, that is, "cross-cutting concerns", are formulated already in the requirements, and therefore can be dealt with in a more controlled manner in the "earlier" phase of software architecture design.

We use Fiadeiro and Maibaum's [11] precise concept of software architectures organised as diagrams over a category of component specifications, where architecture semantics is defined as a colimit specification. The diagram structure suggests aspect introduction via an appropriate variant of graph transformation. Single-pushout rewriting in categories of total homomorphisms has previously already been used for different kinds of "enrichment" transformations; we identify "zigzag-path homomorphisms" as producing a category where many practically useful aspect introductions turn out to be such single-pushout transformations, and present the relevant theorems concerning pushout existence and pushout construction.

Practical aspect introduction (e.g., privacy) always breaks some properties (e.g., "message can be read in transit"); therefore, aspect introduction transformations cannot be designed to be semantics preserving. Our special categorical setting enables selective reasoning about property preservation in the transformed specifications, and property introduction from the introduced aspects. This method enables us to detect and resolve both conflicts and undesirable emergent behaviors that arise from aspect introduction or interaction.

Keywords

Software architecture, aspect introduction, graph transformation

ME 2017 Sept. 18, 2017, Austin, TX, USA

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

1. Introduction

The relationship between the requirements and the components of a system is not always straightforward. Regardless of the size of the system, sometimes in practice a single requirement might not be implemented by a single component and some components might implement more than one requirements [26]. This is because functional decomposition "cuts across" other kinds of decomposition.

In spite of the wide acceptance of separation of concerns as a good software engineering principle, most attempts to define concerns try to relate them to programs [26]. "In fact, as discussed by Jacobsen and Ng (2004), concerns are really reflections of the system requirements and priorities of stakeholders in the system" [26]. If we relate concerns with requirements, then aspects are cross-cutting concerns/system concerns that cross-cut through different core concerns or apply to a whole system [18]. Some good examples of aspects are performance, reliability, security, authorization, synchronization, error handling [19, 26].

Aspect-oriented programming [18, 21] is a complementary programming technique to generalized-procedure programming languages ([18]). It allows design and code to be more modular to reflect the developers' view of the system by modularizing away the cross-cutting functionality from the base program into a separate module (also called aspect) [18, 21, 7, 9].

Though the aspect-oriented community claims that AOP has some commendable characteristics, unfortunately, it has some significant drawbacks as well. For the very same reason for which [8] in his famous letter considered goto statements harmful, [3] characterize AOP as the modern-day "OOP goto" and provocatively asks "AOP considered harmful?". Besides that, rather than making the system simpler, sometimes AOP may increase the complexity of the system (to a certain degree) and lead to almost untraceable problems [21]. However, not only the methodology, but also having to deal with aspects at the programming language level, is one of the reasons that inevitably leads AOP to many of these problems.

We can mitigate the problems in aspect-oriented software development by introducing aspects in the earlier stages of software development, in particular at the software architecture level (a comprehensible higher level abstraction of an overall system structure).

Introducing aspects at the architecture level streamlines the process of aspect-oriented software development and has some other potential advantages. Our methodology mitigates the complexity of system evolution by making the system evolution mechanical instead of manual. Aspects at the architecture level allow developers and other stakeholders to recognize, represent, analyze and evaluate its abstract representation at the earlier stages of software development. As a consequence, the system representation at this level is more comprehensive/all inclusive, and the design decisions will be clearly captured in the actual code. It also induces some other benefits in terms of documentation (i.e., user, system, and design documentation) and cost. The technique we are using to define the system specification from component and connector specifications will allow us not only to reuse small components but also the whole system, i.e., make it a part of a larger system. Finally, we will be able to analyze the new architecture by proving its safety and liveness properties, check its conformance with the old architecture, and detect conflicts, if any exists, due to feature interactions.

In this paper, we address the following problem: "How to introduce aspects at the software architecture level?". Our solution is to develop a technique to deal with aspects by encapsulating them as graph transformation rules on the diagram representing the architecture and applying the aspect by performing the graph transformation. In order to implement our solution, we need to answer the following challenging question:

Research Question (How to Introduce Aspects). What transformation technique will allow us to introduce aspects at the architecture level and verify the properties that need to be preserved from the old architecture and the properties introduced by the aspect.

An effective solution to our research question depends on a few considerations, such as how do we specify the components and connectors, and how do we define the architecture or system specification. [11, 10] introduce a logic to describe (specify) architecture components and connectors and a technique to specify the system or architecture specification as a diagram involving component and connector specifications. The logic they introduce is similar to Modal Dynamic Logic, except that here actions are propositions. The language has a special logical principle called "locality". This notion of "locality" has been used successfully to represent the software engineering principle of data abstraction, scope and encapsulation. In order to combine the components, the category theoretic (CT) notion of colimit is used. The technique used to specify the system specification from components and connectors is independent of the underlying logic and successfully modeled software engineering principles such as modularity, inheritance, incrementality, reusability, and other related concepts [11, 12, 13]. Therefore diagram transformation implies aspect introduction.

Since none of the present graph transformation approaches (i.e., double-pushout, single-pushout, hyperedge replacement graph grammars) are applicable in our case, in order to perform the graph transformation, we are proposing a new transformation technique. The technique is inspired by the hyperedge replacement graph transformation technique. Besides that, the category-theoretic notions Kleisli category, monad and pushout also have some contribution to define the transformation and finally, help us to figure out a structured way to introduce an aspect at the architecture level by performing a transformation.

This paper will proceed as follows: In Section 2 we have briefly introduced the logic and formally defined some of the essential terminologies as a prerequisite for a better understanding of our research goal. Then, in Section 3, we have illustrated our research challenge, analyzed the application of the established graph transformation techniques in our setting, followed by Section 4; the formal representation of our zigzag transformation methodology and terminologies. Section 5 contains two graph transformation rules and an example of aspect introduction. How to construct the resulting architecture by applying our transformation technique are explained in section 6. In Section 8, we have mentioned some related work along with the similarities and dissimilarities between different approaches. The conformance check of the new system architecture to the old system architecture is introduced in Section 7. Finally, in Section 9, we have mentioned our future work followed by the conclusions in Section 10.

2. Formal Definition of System Architectures

The logic we are using was introduced by Fiadeiro and Maibaum [11]. The *specification* of a component is a pair (θ, Φ) where θ is the signature of the component and Φ is a finite set of formulae over θ.

A *signature homomorphism* σ from θ_1 to θ_2 identifies the symbols in θ_2 that correspond to the symbols in θ_1.

A *specification homomorphism* between two component specifications (θ_1, Φ_1) and (θ_2, Φ_2) is a signature homomorphism σ from θ_1 to θ_2 such that for every axiom $p \in \Phi_1$ and also for the locality axiom [11] which asserts a useful kind of local control for the state transition semantics of θ_1, the translation $\sigma(p)$ is a semantic consequence of Φ_2.

Fact 1. [11, Prop. 3.1.7] *Component specifications and their specification morphisms form a category SPEC.*

Following Fiadeiro and Maibaum [11], we use colimits to define the semantics of *system architectures*, which are defined to be diagrams over this category. A diagram over a category is a "shape graph" together with an assignment of objects to vertices, and of compatible morphisms to edges — the following definitions follow standard practice:

Definition 2. A *(directed) graph* $\mathscr{G} = (\mathscr{V}, \mathscr{E}, src, tgt)$ consists of a vertex set $\mathscr{V}$, and edge set $\mathscr{E}$, and the source and target mappings $src, tgt : \mathscr{E} \to \mathscr{V}$.

Definition 3. A *diagram* in a category $\mathscr{C}$ is a graph homomorphism $\mathscr{D} : \mathscr{I} \to |\mathscr{C}|$ for some graph $\mathscr{I}$. The graph $\mathscr{I}$ is called the *shape graph* of the diagram [2].

In the following, we assume a choice of colimits in *SPEC*, and follow [11] in referring to the logical consequences of the axioms of a specification as its *properties*:

Definition 4. A *system architecture* is a diagram in the category *SPEC*.

The *properties* of a system architecture are the properties of the colimit of the diagram.

In the view that considers diagrams as functors instead of just graph homomorphisms, the standard homomorphisms concept for these diagrams is just that of natural transformations:

Definition 5. A *system architecture homomorphism* $\mathscr{H} : \mathscr{A}_1 \to \mathscr{A}_2$ from architecture $\mathscr{A}_1$ to architecture $\mathscr{A}_2$ is a triple $\mathscr{H} : (\mathscr{H}_\mathscr{V}, \mathscr{H}_\mathscr{E}, \mathscr{H}_{SpecMap})$ consisting of:

- node mapping $\mathscr{H}_\mathscr{V} : \mathscr{A}_1.\mathscr{V} \to \mathscr{A}_2.\mathscr{V}$,
- edge mapping $\mathscr{H}_\mathscr{E} : \mathscr{A}_1.\mathscr{E} \to \mathscr{A}_2.\mathscr{E}$,
- transformation $\mathscr{H}_{SpecMap}$ selecting for each node $n_1 : \mathscr{A}_1.\mathscr{V}$ a specification homomorphism

$$\mathscr{H}_{SpecMap}\, n_1 : SpecHom\ (\mathscr{A}_1\ n_1)\ (\mathscr{A}_2\ (\mathscr{H}_\mathscr{V}\ n_1)),$$

such that for each edge $e : \mathscr{A}_1.\mathscr{E}$ we have the following:

$$\begin{aligned} \mathscr{A}_2.src(\mathscr{H}_\mathscr{E}\ e) &= \mathscr{H}_\mathscr{V}(\mathscr{A}_1.src\ e) \\ \mathscr{A}_2.tgt(\mathscr{H}_\mathscr{E}\ e) &= \mathscr{H}_\mathscr{V}(\mathscr{A}_1.tgt\ e) \\ \mathscr{H}_{SpecMap}(tgt\ e) \circ \mathscr{A}_1\ e &= \mathscr{A}_2\ (\mathscr{H}_\mathscr{E}\ e) \circ \mathscr{H}_{SpecMap}(src\ e) \\ \mathscr{H}_{SpecMap}(src\ e) & \text{ is an isomorphism} \end{aligned}$$

Definition 6. *SysArchs* is the category where objects are system architectures and morphisms are system architecture homomorphisms.

The last condition of Def. 5 is sufficient to ensure that *SysArchs* has pushouts.

With this concept of system architecture in hand, introducing aspects at the system architecture level is now easily understood as a kind of diagram transformation.

3. Aspect Introduction as Diagram Transformation

For the system architectures as defined in the previous section, we now explore how aspect introduction can be performed via diagram transformation; in the current section we concentrate on the shape graph aspect of this. By considering a concrete example, we will demonstrate that the kind of aspect introduction we aim for is not covered by existing graph transformation concepts; in the following sections, we will then find an appropriate formalization for the kind of diagram transformation we need.

Literally, transformation of something means change in its shape or appearance. In graph transformation, the underlying entity whose form is changed is a graph, and this change is controlled by rules $r = (\mathcal{L} \to \mathcal{R})$ (often called production rules). The graph on which we apply the transformation rule is called the application graph ($\mathcal{A}$). Rule application to some application graph ($\mathcal{A}$) requires finding an occurrence of an instance of a left-hand side ($\mathcal{L}$). The graph that is the outcome of the transformation is called the result graph ($\mathcal{B}$).

Graph transformation is a powerful tool and it can be used to resemble all the common terminologies associated with aspect oriented programming. The term "aspect" has the same meaning for both AOP and the graph transformation technique, i.e., cross-cutting concerns. A graph rewriting *rule* can be considered as an *aspect program*. The *join point* and *advice* of AOP can be represented as the *left-hand side* and *right-hand side* of a *rule* in the graph transformation technique. The *matching* of a left-hand side into the application graph corresponds to the term *pointcut*. The way a *transformation step* is generated is equivalent to *weaving*, and the combination of both the base and aspect programs is resembled by the *result graph*.

The primary motivation for our work is to streamline the process of aspect-oriented software development by developing a technique to introduce aspects in the early stages of software development, i.e., at the software architecture stage. To recall, *aspects* are concerns (priorities of stakeholders) that cross-cut through different core/functional concerns. Through the following example, we will illustrate what we mean by *aspect introduction*.

Consider the diagram in Figure 1. This diagram is an architecture of sender-receiver communication. Here the com-

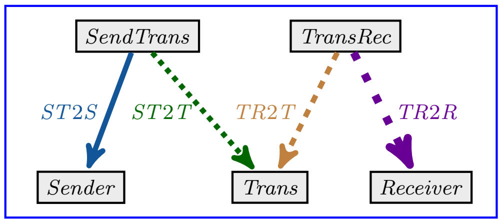

Fig. 1: Unsecured communication

ponent *Sender* sends a message to the component transmitter (*Trans*), and the transmitter transmits it to the component *Receiver*. In order to synchronize/communicate, *Sender* and *Trans* share the sub-component *SendTrans*. The connection via this sub-component along with the two arrows *ST2S, ST2T* identifies the commonalities between *Sender* and *Trans*, and allows them to communicate. Similarly, the components *Trans* and *Receiver* synchronize by sharing the connector $\xleftarrow{TR2T} TransRec \xrightarrow{TR2R}$.

Now, consider the architecture in Figure 2. Here the component *Sender* sends an enciphered message to the component transmitter (*Trans*), and the transmitter transmits it to the component *Receiver*, but the *Receiver* deciphers the message before its final acceptance.

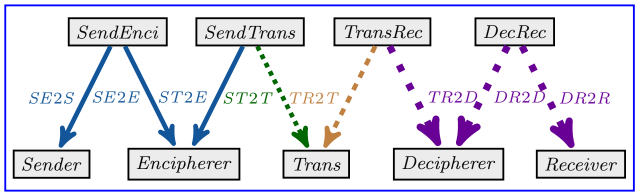

Fig. 2: Secured communication

So, what did we do here? We introduced a security aspect into an architecture where unsecured communication existed and made the architecture "secured". But the challenging question is: How can we systematically introduce such aspects into software architectures? Simple cases of this, as the one shown in Figure 3, suggest that standard categorical graph transformation concepts, such as the double-pushout (DPO) approach [4], should be applicable. Different style and color edges are used in the diagrams to make the matching obvious.

However, with more complex application architectures, we find situations where we consider aspect introduction to still make sense, even though no total shape graph homomorphism exists from the left-hand side of the rule, and such total *matchings* would be required both for the single-pushout approach and for the double-pushout approach to graph transformation.

For example, given an architecture containing a "secure" communication channel, if we want to introduce reliability

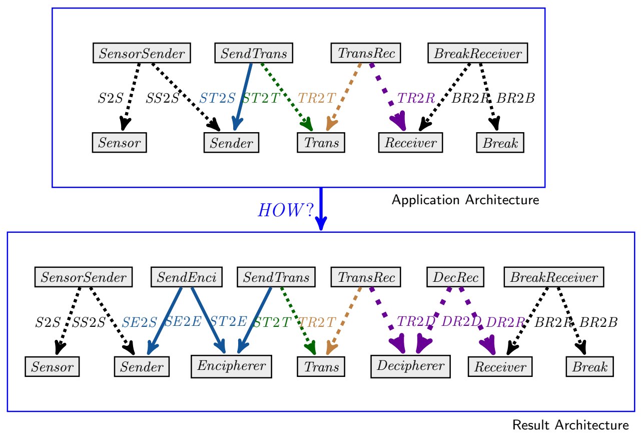

Fig. 3: Introducing communication security

into this architecture through the DPO approach and make the architecture both secure and reliable, the transformation would have to look like Figure 4.

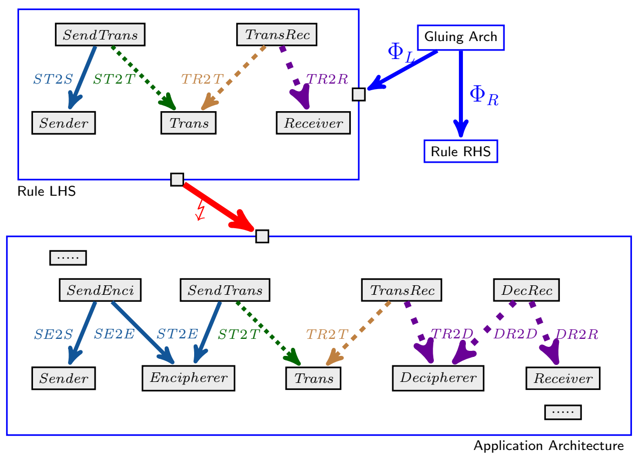

Fig. 4: Adding reliability to secure communication

However, we notice that there is no structure-preserving matching $\mathcal{X}_L$ from the "Rule LHS" to the "Application Architecture", since we cannot find in the "Application Architecture" any match for the edges *ST2S* and *TR2R* of the LHS. Hence, in this kind of setting, the conventional graph homomorphism "does not work", and therefore the DPO and SPO approaches cannot be used directly.

The type of matching we require is pictured in Figure 5; the fact that we need this kind of "indirect" matching is the most obvious reason why conventional DPO/SPO is not directly applicable. Here, a single edge can map to a *zigzag paths* and mapping between vertices are *specification homomorphism*. For further illustration see section 4.

4. Zigzag Path Homomorphisms

Although we are using directed graphs to underpin our system architectures, the paths we will be considering for modification by aspect introduction are not directed[1], that is, edges in such paths can be traversed in any direction:

[1]A *directed path* is a sequence of vertices connected by directed edges where all the edges are traversed along their direction.

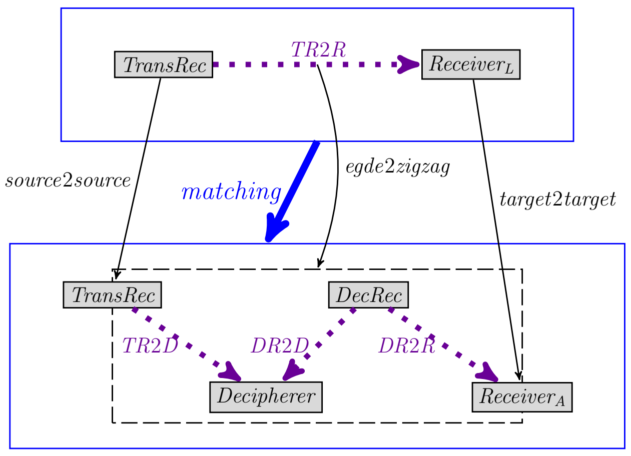

Fig. 5: Matching single edge to undirected path

DEFINITION 7. A *zigzag path* p in a graph $\mathscr{G}$ is a alternating list $p = \langle v_0, e_1, v_1, e_2, v_2, \ldots\ldots e_k, v_k \rangle$ of vertices and directed edges traversed in arbitrary directions, where

- the first and last element are always vertices, called the *source* and the *target* of the path;
- $k \geq 0$ is the *length* of p;
- e_i is incident with v_{i-1} and v_i for $i \in 1\ldots, k-1$.

The set of all zigzag paths in $\mathscr{G}$ will be written $\mathscr{Z}path_{\mathscr{G}}$.

DEFINITION 8. Given two graphs $\mathscr{G}_i = (\mathscr{V}_i, \mathscr{E}_i, src_i, tgt_i)$ for $i = 1, 2$, a $\mathscr{Z}path$ homomorphism $\mathscr{H} : \mathscr{G}_1 \to \mathscr{G}_2$ consists of two functions, $\mathscr{H}_v : \mathscr{V}_1 \to \mathscr{V}_2$, and $\mathscr{H}_{ep} : \mathscr{E}_1 \to \mathscr{Z}path_{\mathscr{G}_2}$ such that the two diagrams in Figure 6 commute.

$$\begin{tikzcd}
\mathscr{E}_1 \arrow[r, "\mathscr{H}_{ep}"] \arrow[d, "src_1"] & \mathscr{Z}path_{\mathscr{G}_2} \arrow[d, "source_2"'] & \mathscr{E}_1 \arrow[r, "\mathscr{H}_{ep}"] \arrow[d, "tgt_1"] & \mathscr{Z}path_{\mathscr{G}_2} \arrow[d, "target_2"'] \\
\mathscr{V}_1 \arrow[r, "\mathscr{H}_v"] & \mathscr{V}_2 & \mathscr{V}_1 \arrow[r, "\mathscr{H}_v"] & \mathscr{V}_2
\end{tikzcd}$$

Fig. 6: $\mathscr{Z}path$ Graph Homomorphism

For the matchings as explained in the previous section, we now allow edges to be matched to zigzag paths. In the example there, this corresponds to identifying the zigzag path *SendTrans* $\to$ *Encipherer* $\leftarrow$ *SendEnci* $\to$ *Sender* in the secured communication architecture in Figure 4 as "reasonably matching" the communication setup *ST2S* : *SendTrans* $\to$ *Sender* in the rule LHS.

DEFINITION 9. A *system architecture Zpath homomorphism* is a tuple $\mathscr{H} = (\mathscr{H}_{\mathscr{V}}, \mathscr{H}_{\mathscr{Z}}, \mathscr{H}_{SpecMap})$ consisting of a shape graph $\mathscr{Z}path$ homomorphism $(\mathscr{H}_{\mathscr{V}}, \mathscr{H}_{\mathscr{Z}})$, and the mapping $\mathscr{H}_{SpecMap}$ that assigns to each vertex $n : \mathscr{V}_1$ a specification homomorphism $\mathscr{H}_{SpecMap}\ n : \mathscr{A}_1\ n \to \mathscr{A}_2\ (\mathscr{H}_{\mathscr{V}}\ n)$, such that source node specifications are preserved, that is, for each edge $e : \mathscr{A}_1.\mathscr{E}$, the specification homomorphism $\mathscr{H}_{SpecMap}(src\ e)$ is an isomorphism.

Note that we included no conditions on interaction between the range of $\mathscr{H}_{SpecMap}$ and the specification homomorphisms labelling the edges of the image paths of $\mathscr{H}_{\mathscr{Z}}$ — for different kinds of property preservation, different and rather specialised conditions are necessary, which are beyond the scope of the current paper.

DEFINITION 10. We define *ZpathGraphs* is the category where objects are graphs and morphisms are Zpath graph homomorphisms.

There is an obvious embedding functor from *Graphs* into *ZpathGraphs*, and this preserves pushouts:

THEOREM 11. *For every span in Graphs, a pushout in Graphs for that span is also a pushout in ZpathGraphs.*

For defining graph transformations, we now move to the category that allows simple connections in architectures to be matched to zigzag paths over several components:

DEFINITION 12. *SysArchsZ* is the category where objects are system architectures and whose morphisms are system architecture Zpath homomorphisms.

Theorem 11 carries over to this category:

THEOREM 13. *For every span in SysArchs, a pushout in SysArchs for that span is also a pushout in SysArchsZ.*

5. Aspect Introduction Rules

Using the well-known example of sender-receiver communication [1] that we already employed for the examples in section 3, we will now illustrate our *Zpath graph transformation* based technique.

DEFINITION 14. A *Zpath architecture transformation rule* consists of two system architectures L and R connected by a single Zpath system architecture homomorphism $\Phi : L \to R$.

Such a rule can be *applied* to system architecture A via the *matching* Zpath system architecture homomorphism $\Xi : L \to A$ with *result* system architecture B if a pushout in *SysArchZ* of the following shape exists:

$$\begin{tikzcd} L \arrow[r, "\Phi"] \arrow[d, "\Xi"'] & R \arrow[d, "X"'] \\ A \arrow[r, "\Psi"] & B \end{tikzcd}$$

Such a pushout is also called a *Zpath architecture transformation step* from A to B via the rule $\Phi : L \to R$.

Security Introduction:

For any given architecture where sender-receiver communication exists, if we want to introduce security into it by performing a Zpath graph transformation, the transformation rule we will apply is shown in Figure 7. In fact, transformation via this rule explains the security introduction shown in Figure 3.

Reliability Introduction:

For any given architecture where sender-receiver communication exists, the rule for introducing reliability into it is depicted in Figure 8. This is the rule we “tried to apply” in Figure 4, where the right-hand-side is not drawn.

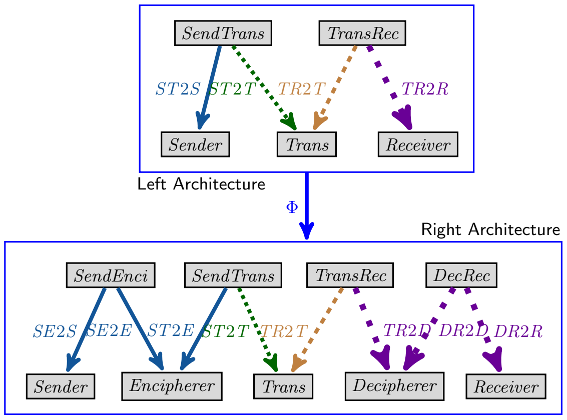

Fig. 7: Rule: Security Introduction

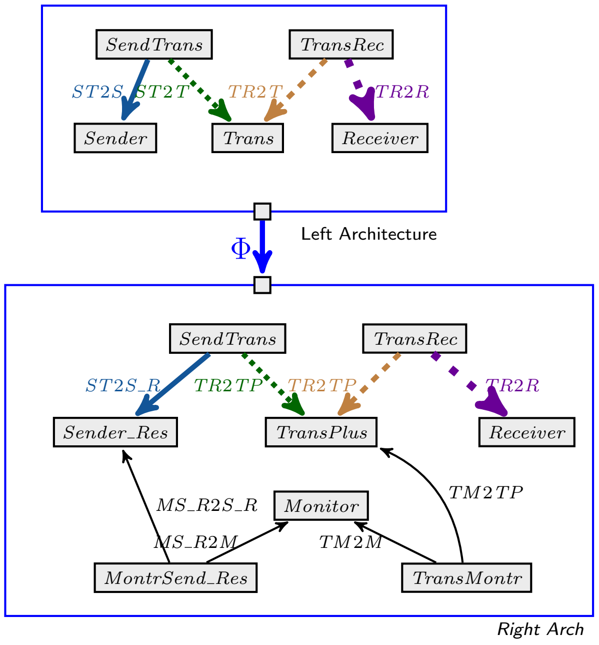

Fig. 8: Rule: Reliability Introduction

Introduce Reliability on Secure Communication

Consider we have a secured communication architecture as application graph, and we wish to introduce reliability on top of this to make the architecture both secured and reliable. Applying the rule of Figure 8 succeeds, since this pushout in *SysArchZ* exists; this is shown in Figure 9.

However, *SysArchZ* does not have all pushouts, so that aspect introduction via a chosen rule and matching may be impossible (if no commuting square completing this span exists), or may require additional design decisions (if no single “least” such commuting square exists).

Nevertheless, even if both rule and matching map LHS edges to non-trivial zigzag paths, *SysArchZ* pushouts still exist, as we investigate in more detail in the next section.

6. Amalgamating *SysArchZ* Pushouts

Although *SysArchZ* does not have all pushouts, many aspect introduction rule applications are still possible, and can be completed via pushouts, due to the typical shape of as-

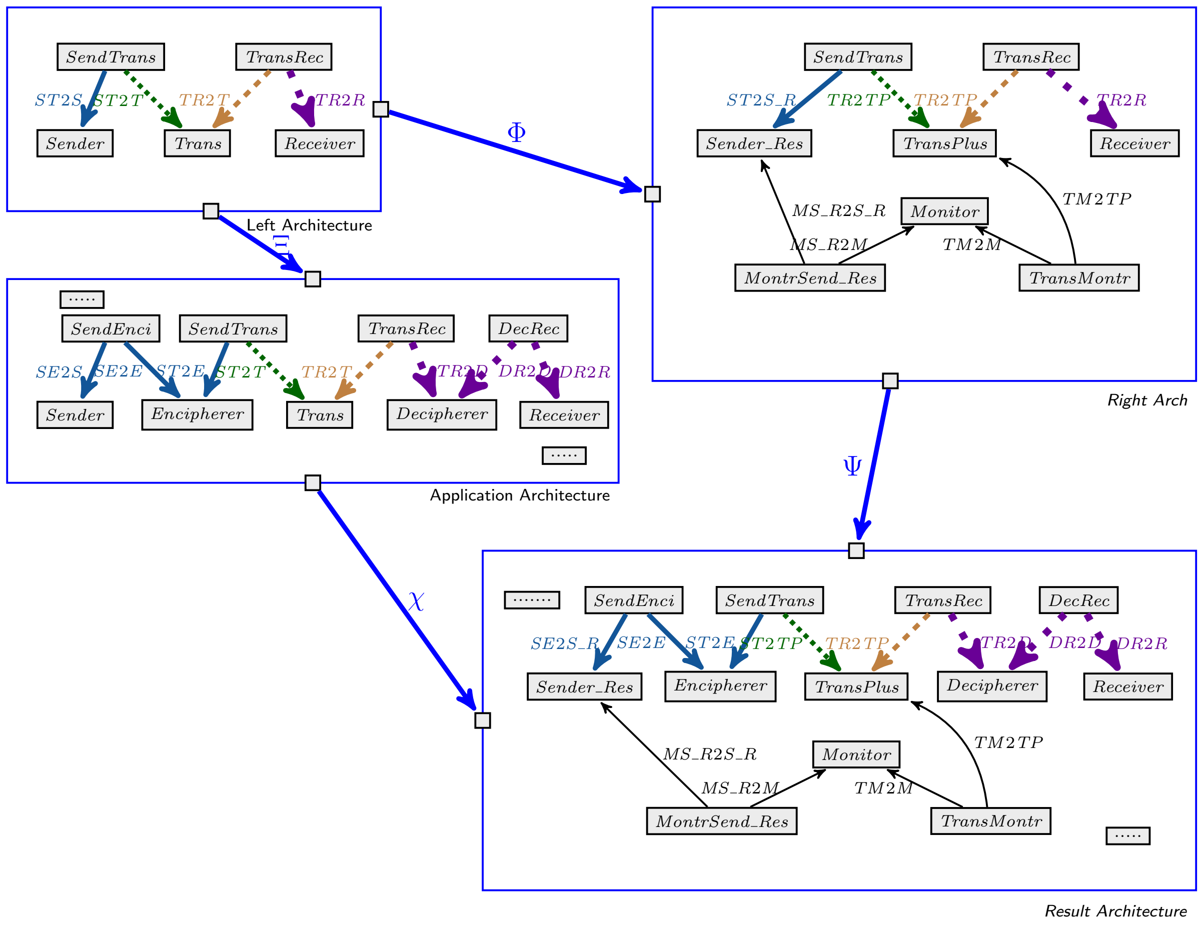

Fig. 9: Applying Reliability Introduction to Secured Communication via Zigzag Matching

pect introduction rules that can be observed already in the examples provided so far: The left-hand side is usually a single zigzag path, and some of the edges of the LHS are replaced with zigzag paths on the RHS, while other LHS edges are preserved. These preserved edges can be matched to zigzag paths without creating conflicts — technically, without creating a rule-matching-span that has no pushout.

The general reason for this is that *SysArchZ* pushouts can be amalgamated along *SysArch* pushouts.

Theorem 15. *If the span $\mathscr{A} \leftarrow \mathscr{L} \rightarrow \mathscr{R}$ in SysArchZ can be factored via three pushouts in SysArchs as shown in Figure 10, and if the two SysArchsZ spans $\mathscr{A}_1 \leftarrow \mathscr{L}_1 \rightarrow \mathscr{R}_1$ and $\mathscr{A}_2 \leftarrow \mathscr{L}_2 \rightarrow \mathscr{R}_2$ have pushouts in SysArchsZ, then $\mathscr{A} \leftarrow \mathscr{L} \rightarrow \mathscr{R}$ has a pushout in SysArchsZ, too.*

This amalgamation theorem allows us to decompose prospective rule application pushouts of rules like those of Figs. 7 and 8 into little pieces induced by the subgraphs of the left-hand side induced by single edges, or by node sets. We discuss the different kinds of these pieces in Sects. 6.1–6.3.

The situation is further simplified if we restrict ourselves to architectures with *connector-component bipartition*, where each node of the shape graph either is a “connector” that has only outgoing edges, or is a “component” that has only incoming edges. For system architecture (Z-path) homomorphisms, we then restrict the specification homomorphisms associated with source nodes of edges in the source diagram

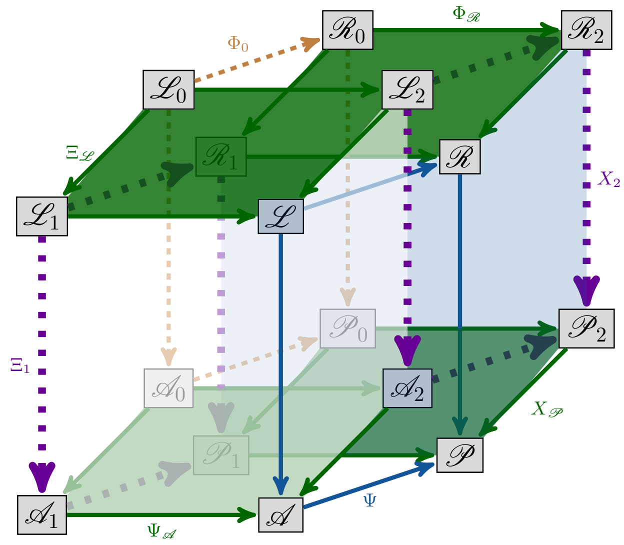

Fig. 10: Amalgamation Theorem

to be isomorphisms. In this paper, for the sake of simplicity we strengthen this restriction further to only allow identity homomorphisms. For example, *source2source* in Figure 5 has to be an identity.

6.1 Discrete LHS

Working with subgraphs induced by node sets of the left-hand side is necessary for separating the context of a rule application from the modifications introduced by the rule, and also for parts of the RHS that are not in the (zigzag-) image of the LHS, as for example the monitor components of the reliability introduction rules of Figure 8, for which we show the corresponding rule fragment in Figure 11.

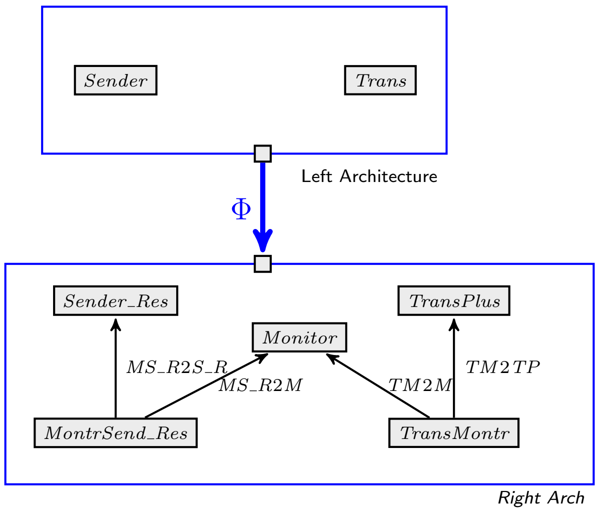

Fig. 11: Discrete-LHS Fragment of Reliab. Intro.

In the situations we encountered so far, the following (rather obvious) theorem is sufficient, although it excludes context that is attached to connectors:

THEOREM 16. *A* SysArchsZ *span* $A \xleftarrow{\Xi} L \xrightarrow{\Phi} R$ *where* L *is discrete (no edges) and both* Ξ *and* Φ *do not map any node of* L *to a source node of an edge in* A *respectively* R *always has a pushout in* SysArchsZ.

6.2 Unproblematic Single-Edge LHS

If a single-edge LHS is mapped via a non-zigzag homomorphism containing only identity specification homomorphism for the components, a pushout obviously always exists, as indicated in Figure 12.

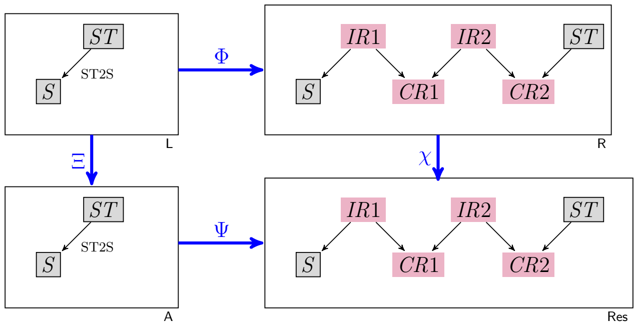

Fig. 12: Component addition / identity matching

If we restrict both rule and matching to non-zigzag homomorphisms, but allow the target of the LHS edge to be mapped with arbitrary specification homomorphisms on both sides, we get an architecture pushout where that target is assigned the corresponding specification pushout, as sketched in Figure 13.

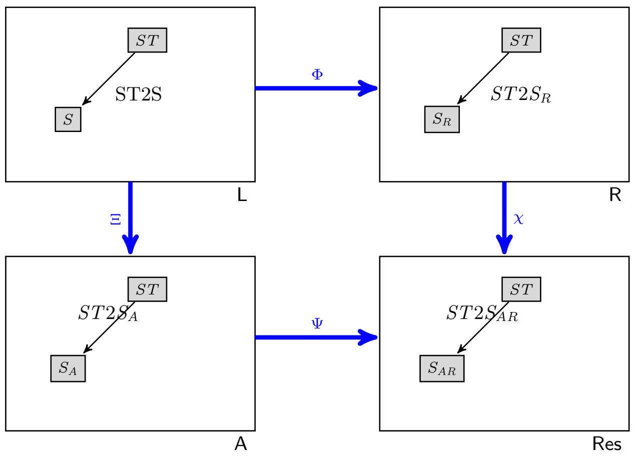

Fig. 13: Component expansion both ways

These two cases can actually be combined, as depicted in Figure 14:

THEOREM 17. *A* SysArchsZ *span* $A \xleftarrow{\Xi} L \xrightarrow{\Phi} R$ *where*

- *the shape of* L *is* $\bullet \rightarrow \bullet$,
- *one of* Ξ *and* Φ *is a non-zigzag homomorphism,*
- *the source node of the edge in* L *is associated with identity specification homomorphisms in both* Ξ *and* Φ,

always has a pushout in SysArchsZ.

(The identity specification homomorphisms can be replaced with isomorphisms, which however would make the drawings more confusing.)

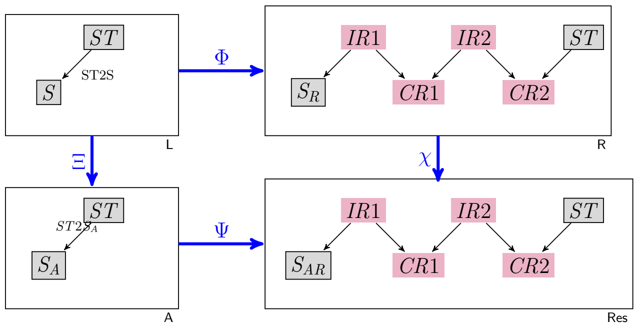

Fig. 14: Component addition / component expansion

For the above scenario in Figure 14, the component S_{AR} is a pushout construct of the span $S_A \xleftarrow{\Xi} S \xrightarrow{\Phi} S_R$. Other components, i.e., CR_1,CR_2 are direct copies of the preimages from the right architecture. All the connectors, i.e., IR_1,IR_2,ST in the result architecture are also the direct copies of their preimages.

6.3 Ambiguous Matchings

Now, let us say, an edge is relaxed in both the Application and the Right hand side graph (relax-relax) by adding a couple of components and connectors and we want to contract them to a single zigzag path. If we preserve the *connector-component alternating* pattern and consider the mapping between architectures as a System Architecture Zpath Homomorphism then there are only two possibilities to contract them to a single zigzag path with the minimal components and connectors such that the above diagram commutes. These two possibilities are as follows:

1. Disconnect the *target* of the Application architecture and the *source* of the Right hand-side architecture and glue them to a single zigzag path (Res-2).
2. Disconnect the *source* of the Application architecture and the *target* of the Right hand-side architecture and glue them together (Res-1).

So, in this scenario, two design choices are available and the result architecture varies depending on the design decision (choice) one makes.

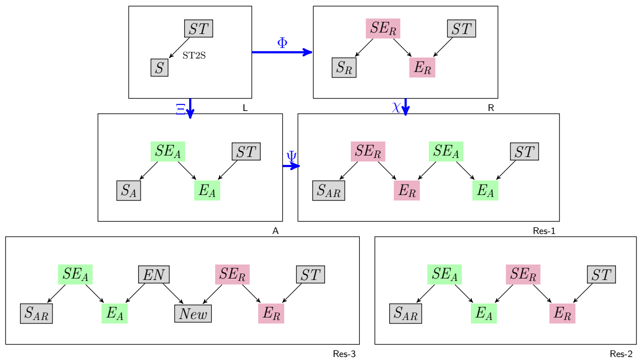

Fig. 15: Component addition both ways

Whether this transformation is possible or not depends on the relationship between the connectors associated to establish a connection (in the result architecture) between the application and the right hand-side architecture. A connection is established in the result architecture by breaking two connections, one each for the application and the right hand-side architecture. In the above example, for Res-1 the connection $SE_A \to S_A$ in the application architecture and $ST \to E_R$ in the right hand side architecture got disconnected to establish a connection $SE_A to E_R$ in the result architecture. This transformation is obvious if a specification homomorphism $SE_A \to ST$ exists. In the other case, i.e., Res-2, the obvious transformation depends on the existence of the specification homomorphism $SE \to ST$. If neither of the homomorphisms exists, then we have to introduce new components and connectors to let the transformation take place and one of this scenario is pictured by Res-3 of the figure 15.

Theorem 18. *For "*both-relax*" case spans, the* SysArchsZ *category does not have any pushouts.*

We have two potential pushout candidates with minimal components and connectors, but neither of them is a pushout object. There are system architecture homomorphisms that exist between them but they are not unique up to isomorphism. In reality these two architectures could be completely different. Though we do not have a pushout, if we select/consider one of our design decisions, then the construction of the result architecture would be systematic and it could be one of the pushout candidates.

6.4 User-Guided Rule Application

As far as pushouts can be constructed for single edges as discussed above, these can be amalgamated for rules with more complex left-hand sides due to Theorem 15.

For a defined set of production rules (e.g., security introduction, reliability introduction) and given an application architecture, if we apply our transformation technique, in all cases except one (Sect. 6.3), the result architecture we obtain after the transformation is a pushout object. If an aspect introduction matches an LHS-edge to a zigzag path with new components in the right-hand side architecture, and the matching of the edge into the application architecture is also a zigzag path, then the category *SysArchsZ* does not have a pushout for the span consisting of that rule with that matching. In this case, the transformation is semi-automatic. The designer will have to make design decisions, which may result in different desirable and undesirable properties becoming valid for the transformation result.

7. Conformance Check

One of the great advantages of our research work is that it makes the analysis of the system architecture properties feasible.

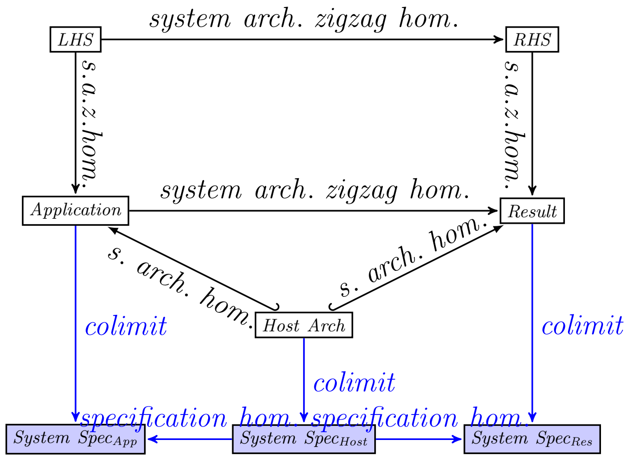

Fig. 16: System Architecture and System Specification

Conformance of the result system architecture with the application system architecture is not straight-forward. Figure 16 illustrates this statement: A system architecture Zpath homomorphism between two system architectures, e.g., $Application \xrightarrow{S.A.Z.Hom.} Result$, does not necessarily imply the existence of a specification homomorphism between the colimits of these diagrams. Proving properties of the result system architecture to check its conformance with the application system architecture is not exhaustive either. Depending on different scenarios, how aspect introduction modifies an edge, in some cases we can systematically (even automatically) check the conformance of the new result system architecture to the old application system architecture without proving any proof obligations. Propagation of RHS properties into the result architecture, also conforms to the same scenario.

8. Related Work

The paradigm of *aspect-oriented software development* (AOSD) first appeared at the University of Twente in the Netherlands at the code level. However, work in aspects is no longer limited to the implementation phase of the software development. Over the last decade, the AOSD community has tried to transfer this idea into earlier phases of the software development life cycle; namely in domain analysis, requirement analysis, architecture design, and modeling.

The modeling community is doing a large amount of work to weave aspects in models, specifically, UML models [17, 27, 23, 22, 15, 14, 25, 5, 16, 28, 24]. Since model weaving is a special case of transformation; some interesting works in weaving aspects in models are explained in the following sections along with their similarities and dissimilarities with our work.

Whittle and Jayaraman [28] developed an aspect-oriented modeling tool MATA (Modeling Aspects Using a Transformation Approach) that uses an existing graph transformation technique over the concrete syntax of the UML modeling language to weave aspects. In order to write a graph rule, rather than using general *LHS* and *RHS*, they defined three stereotypes, i.e., create, delete, and context (unchanged), (similar to the approach applied in VIATRA developed by Csertán *et al.* [6]) which allowed them to write a rule on a single model instead of repeating unchanged elements in both the *LHS* and the *RHS*.

Morin *et al.* [24] worked on a generic AOM approach called GeKo (generic composition with Kermeta) to weave aspects into any model with a well-defined metamodel. Here, two models, the *base* and the *advice* are weaved with the help of a third model and two morphisms. The third mode is called the *pointcut*, and the two morphisms are defined from the *pointcut* to the *base* and the *advice* respectively. The morphisms prescribe the deletion, preservation/edition, and addition. This weaving process is similar to defining a rule in the Double-pushout graph transformation approach explained in [4].

The work we are doing is to introduce aspects at the earlier phase of the software development life-cycle, i.e., in the architecture level, by performing a graph transformation. Although in terms of goals, we have some similarity with the work of the AOM community, our approach is completely different from their approaches. The way [20, 24] explained *aspects* contradicts their traditional definition. A couple of vital points that make our method divergent from AOM weaving are the following:

- Weaving is not a general transformation; it is a special type of transformation. It is usually a non-automatic laborious operation where both *base* and *aspect* models get composed to get a weaved model. In contrast, a transformation is automatic where predefined rules are applied to a bigger application system.

- Most of the above AOM approaches claim that they can detect conflicts (unavailability to weave an *aspect* with a *woven base*) and resolve them by sequencing aspects or changing the rules. But the way they define the conflict does not work for system architecture. After introducing an aspect to an application architecture, further aspect introduction to the resultant architecture by general weaving/transformation technique might be unavailable to some of the re-defined rules though neither the rules nor the resultant architecture is wrong.

One of the potential solutions to this problem is to introduce a new transformation technique by keeping the nature of aspect introductions in mind. Our zpath graph transformation technique is capable of addressing this issue. Besides that, it allows us to systematically verify the conformance (property preservations) of the old system architecture to the new.

9. Future Work

Besides identifying the "Zigzag-path homomorphisms" and elaborating the "Zigzag transformation", we have provided some meta-theorems that make it (sometimes) unnecessary to re-prove properties for transformation results, or make it easier to obtain result properties from component and aspect properties. From our example, we have experienced that proving the well-definedness and validating the properties of system architecture is a tedious redundant job. So, we are currently developing some tool support that will make our methodology mechanical and make our evaluation and validation process feasible. Since our methodology is independent of the underlying logic, one of the potential future steps would be application of the theory in industrial settings by applying widely used architecture languages, e.g., AADL,EAST-ADL.

10. Conclusion

Working with aspect introduction at the architecture level has many benefits including for documentation, product risk management, understandability, reusability and maintainability. The nature of aspects makes it impossible to apply any of the conventional graph transformation approaches, since those work with exact matchings. Our "Zigzag matching" and the "Zigzag transformation" methodology streamline the process of software system evolution by making aspect introduction in system architecture systematic. Besides, in terms of property preservation, it makes the conformance check of the new system architecture along with the detection and resolution of conflicts and the undesirable emergent behaviours semi-automatic.

11. References

[1] N. Aguirre, T. Maibaum, and P. Alencar. *Extension Morphisms for CommUnity*, pages 173–193. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[2] M. Barr and C. Wells. *Category theory for computing science*, volume 49. Prentice Hall New York, 1990.

[3] C. Constantinides, T. Skotiniotis, and M. Stoerzer. AOP considered harmful. In *1st European Interactive Workshop on Aspect Systems (EIWAS)*, 2004.

[4] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation — Part I: Basic concepts and

double pushout approach. In *Handbook of Graph Grammars*, pages 163–246, 1997.

[5] T. Cottenier, A. Van Den Berg, and T. Elrad. The Motorola WEAVR: Model weaving in a large industrial context. *Aspect-Oriented Software Development (AOSD), Vancouver, Canada*, 32:44, 2007.

[6] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA — visual automated transformations for formal verification and validation of uml models. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 267–270. IEEE, 2002.

[7] J. A. Díaz Pace and M. R. Campo. Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10):66–73, 2001.

[8] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[9] T. Elrad, M. Aksit, G. Kiczales, K. J. Lieberherr, and H. Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001.

[10] J. Fiadeiro and T. Maibaum. Towards object calculi. In *Information Systems| Correctness and Reusability, Workshop IS-CORE*, volume 91, pages 129–178, 1990.

[11] J. Fiadeiro and T. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal aspects of Computing*, 4(3):239–272, 1992.

[12] J. Fiadeiro and T. Maibaum. A mathematical toolbox for the software architect. In *Software Specification and Design, 1996., Proceedings of the 8th International Workshop on*, pages 46–55. IEEE, 1996.

[13] J. L. Fiadeiro and T. Maibaum. Interconnecting formalisms: Supporting modularity, reuse and incrementality. *ACM SIGSOFT Software Engineering Notes*, 20(4):72–80, 1995.

[14] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A generic approach for automatic model composition. In *International Conference on Model Driven Engineering Languages and Systems*, pages 7–15. Springer, 2007.

[15] R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. Providing support for model composition in metamodels. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pages 253–253. IEEE, 2007.

[16] I. Jacobson and P.-W. Ng. *Aspect-oriented software development with use cases*. Addison-Wesley object technology series. Addison-Wesley Professional, 2004.

[17] P. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa. Model composition in product lines and feature interaction detection using critical pair analysis. In *International Conference on Model Driven Engineering Languages and Systems*, pages 151–165. Springer, 2007.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*. Springer, 1997.

[19] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering*, pages 49–58. ACM, 2005.

[20] J. Kienzle, W. Al Abed, and J. Klein. Aspect-oriented multi-view modeling. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 87–98. ACM, 2009.

[21] J. Laukkanen. Aspect-oriented programming, 2008.

[22] B. Morin, O. Barais, and J.-M. Jézéquel. Weaving aspect configurations for managing system variability. In *2nd International Workshop on Variability Modelling of Software-intensive Systems*, 2008.

[23] B. Morin, O. Barais, J.-M. Jézéquel, and R. Ramos. Towards a generic aspect-oriented modeling framework. In *Models and Aspects workshop, at ECOOP 2007*, 2007.

[24] B. Morin, J. Klein, O. Barais, and J.-M. Jézéquel. A generic weaver for supporting product lines. In *Proceedings of the 13th international workshop on Early Aspects*, pages 11–18. ACM, 2008.

[25] C. Siobhan and B. Elisa. Aspect-oriented analysis and design: The theme approach, 2005.

[26] I. Sommerville. *Software Engineering*. International Computer Science Series. Pearson, 2011.

[27] J. Whittle and P. Jayaraman. Mata: A tool for aspect-oriented modeling based on graph transformation. In *International Conference on Model Driven Engineering Languages and Systems*, pages 16–27. Springer, 2007.

[28] J. Whittle and P. Jayaraman. Mata: A tool for aspect-oriented modeling based on graph transformation. In *Models in Software Engineering*, pages 16–27. Springer, 2008.