ASPECT INTRODUCTION THROUGH DIAGRAM TRANSFORMATION

INTRODUCING ASPECTS INTO SOFTWARE ARCHITECTURES BY GRAPH TRANSFORMATION

By MD. NOUR HOSSAIN, M.Sc., B.Sc.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfilment of the Requirements for the Degree Doctor of Philosophy

DOCTOR OF PHILOSOPHY (2018) McMaster University

(Department of Computing & Software) Hamilton, Ontario, Canada.

TITLE: Introducing Aspects into Software Architectures by Graph Transformation

AUTHOR: Md. Nour Hossain

B.Sc. in Computer Science & Engineering,
Rajshahi Unieristy of Science & Technology,
Rajshahi, Bangladesh.

M.Sc in Computer Science,
Brock University,
St. Catharines, Ontario, Canada.

SUPERVISOR: Dr. Tom Maibaum
Dr. Wolfram Kahl

NUMBER OF PAGES: x, 169

Dedicated to my parents, my daughter and my wife

Abstract

While aspect-oriented programming (AOP) addresses the introduction of "aspects" at the code level, we argue that addressing this at the level of software architecture is conceptually and methodologically 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 the precise concept of software architectures organised as diagrams over a category of component specifications, where the architecture semantics are defined as a colimit specification (Fiadeiro and Maibaum, 1992). The diagram structure suggests aspect introduction via an appropriate variant of graph transformation. Single-pushout rewriting in categories of total homomorphisms has already been used previously 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.

We have developed tool support to introduce and analyze aspects at the system architecture level through zigzag graph transformation. The implementation is based on Hets, an initiative of Mossakowski et al. (2007) and consists of two key parts: the language development and the zigzag transformation. The development of the *MFLogic* language is based on the specification language Casl (Astesiano et al., 2002) and uses the logic introduced by Fiadeiro and Maibaum (1992). Besides parsing, syntactic and static semantics correctness checking, the language inclusion in Hets opens the door for automatic property preservation analysis and conflict detection. The main contribution of the tool support in Hets is the automatic aspect introduction and the "result architecture" generation by applying our zigzag graph transformation.

ACKNOWLEDGMENT

Firstly, I would like to express my sincere gratitude to my supervisors Dr. Wolfram Kahl and Dr. Tom Maibaum for their continuous support of my Ph.D. and related research, for their patience, motivation, and immense knowledge. Their guidance helped me throughout my research and writing of this thesis. I could not have imagined having better advisors and mentors for my Ph.D. study.

Beside my advisors, I would like to thank the rest of my thesis committee: Dr. Ridha Khedri and Dr. Jacques Carette for their insightful comments and encouragement, but also for the hard questions which incentivized me to widen my research scope, based on various perspectives.

I thank my fellow colleagues and friends for extending their support in a very special way. I have gained a lot from them, through their personal and scholarly interactions, their suggestions at various points of my research programme.

Additionally, I would like to thank my family: my parents and my brothers and sisters for supporting me spiritually throughout this Ph.D. and my life in general.

And most of all for my loving, supportive, encouraging, and patient wife Ayesha Rouf whose faithful support during all the stages of this Ph.D. is so appreciated. Thank you.

Contents

Abstract iv

Acknowledgement v

1 Introduction, Related Work 1
- 1.1 Motivation . . . 1
- 1.2 Modularization via "Aspects" . . . 1
- 1.3 AOP to the Rescue? . . . 2
- 1.4 Raise the Level of Aspect Introduction . . . 2
- 1.5 Research Question . . . 3
- 1.6 Summary of Contributions . . . 3
- 1.7 Related Work: Aspect Oriented Programming . . . 4
- 1.8 Related Work: Model Transformation . . . 10
- 1.9 Overview of the Thesis . . . 13

2 Background 15
- 2.1 Graphs . . . 15
- 2.2 Category-theoretic Notions . . . 15
- 2.3 Relation-algebraic Notations and Properties . . . 17
- 2.4 Graph Transformation . . . 19
- 2.5 Introduction to MFLogic . . . 22

3 Zigzag Matching for Join Point Patterns 28
- 3.1 Introduction of Running Example (Use Case) . . . 28
- 3.2 Exploration of Established Transformation Techniques . . . 30
- 3.3 Zigzag Path . . . 33
- 3.4 Zigzag Graph Transformation: The Matching . . . 33
- 3.5 The category ZigzagGraphs of graphs . . . 35
- 3.6 Relational Homomorphism . . . 39
- 3.7 Pushout . . . 40

4 Construction of the Resulting Architecture through Zigzag Diagram Transformation 48
- 4.1 Zigzag Graph Transformation . . . 48
- 4.2 Construction of The Result Architecture: All Possibilities . . . 50
- 4.3 Construction of The Result Architecture: Restricted Pushout . . . 54
- 4.4 SysArchsZ Pushout Proofs for Single Edge LHS Cases . . . 57
- 4.5 Component addition both ways: . . . 66

5 Property Preservation **70**
5.1 Case 1: Specification Homomorphism Exists 71
5.2 Case 2: Specification Homomorphism Does Not Exist 72
5.3 Feature Interaction Detection 78

6 Tool **80**
6.1 HETS 80
6.2 CASL 80
6.3 Implementation Details 81
6.4 Installation and Some Other Essentials 89
6.5 How to Use? 94

7 Conclusion and Future Work **100**

Bibliography **102**

Appendix A Amalgamation Theorem and its Proof **107**

Appendix B An Example: Application of Our Methodology **120**
B.1 Architecture of Unsecured-Unreliable Communication 120
B.2 Description of the Secured Communication Architecture 132
B.3 Description of the Reliable Communication Architecture 145
B.4 Graph Transformation Rules 164
B.5 Architecture With Aspect 165

List of Figures

1.7.1 Aspect: security requirements . . . 5
1.7.2 A Simple Drawing Tool (Kiczales, 2006) . . . 6
1.7.3 OO design (Kiczales, 2006) . . . 7
1.7.4 OO program (Kiczales, 2006) . . . 7
1.7.5 OO design with aspect (Kiczales, 2006) . . . 8
1.7.6 OO Scattered Code (Kiczales, 2006) . . . 8
1.7.7 Aspect design (Kiczales, 2006) . . . 9
1.7.8 Aspect program (Kiczales, 2006) . . . 9
1.7.9 Possible error situations with pointcut . . . 10
1.8.1 MATA rule and its application . . . 12
1.8.2 MATA rule and its limitation . . . 14
2.2.1 Commuting cocone . . . 17
2.3.1 Relational graph homomorphism . . . 19
2.4.1 Introduction to Graph Transformation . . . 20
2.4.2 Double-pushout diagram . . . 21
2.4.3 Single-pushout diagram . . . 22
3.1.1 Unsecured communication . . . 28
3.1.2 Secured communication . . . 29
3.1.3 Introducing security in communication . . . 29
3.2.1 Rule: DPO Security Introduction . . . 30
3.2.2 Rule: DPO Reliability Introduction . . . 31
3.2.3 Introduce Reliability by DPO Approach . . . 32
3.2.4 Endeavor to introduce reliability in secured communication by DPO . . . 32
3.4.1 Matching single edge to undirected path . . . 34
3.5.1 Well-definedness of $\mathscr{H}_{13}$ . . . 36
3.5.2 Identity Law . . . 38
3.6.1 Relational zigzag graph homomorphism . . . 39
3.6.2 Relationship between *Graphs*, *RelGraphs*, *RelZigzagGraphs* . . . 40
3.7.1 ZigzagGraphs pushout . . . 40
3.7.2 System Architecture Homomorphism . . . 45
3.7.3 System Architecture Zigzag Homomorphism (Bottom,Up) . . . 46
3.7.4 Identity System Architecture Homomorphism . . . 47
4.1.1 General Modification of a Single Edge . . . 49
4.1.2 Practical Modification of a Single Edge . . . 49
4.2.1 Component expansion both ways . . . 50
4.2.2 Component addition / identity matching . . . 51
4.2.3 Component addition / component expansion . . . 52
4.2.4 Component addition both ways . . . 53
4.3.1 Sample Example: Restricted pushout . . . 54
4.3.2 Amalgamation Theorem: Assumption . . . 55

4.3.3 Amalgamation Theorem: Factoring . . . 56
4.3.4 Discrete-LHS Fragment of Reliab. Intro. . . . 57
4.4.1 Component expansion both ways . . . 58
4.4.2 Pushout in SysArchsZ . . . 58
4.4.3 Diagram in *DESC* . . . 60
4.4.4 Component addition / component expansion . . . 61
4.4.5 Component addition / identity matching . . . 66
4.5.1 Pushout objects and pushout candidates for Figure 4.2.4 . . . 67
4.5.2 *AllRelax* case is *SysArchsZ* has no pushout . . . 68
4.5.3 AllRelax case is SysArchsZ has no pushout . . . 69
5.0.1 System Architecture and System Specification . . . 70
5.1.1 System Architecture and System Specification . . . 71
5.2.1 Host System Architecture and System Specification Homomorphism . 74
5.2.2 Glue System Architecture and System Specification Homomorphism . 75
5.3.1 Reliable-Secured Communication . . . 78
A.0.1Amalgamation Theorem: Assumption . . . 107
A.0.2Amalgamation Theorem: Factoring . . . 108
A.0.3Amalgamation Theorem: Auxiliary Part . . . 111
A.0.4Amalgamation Theorem: Universal Property . . . 114
B.1.1Unsecured-Unreliable Communication Architecture . . . 120
B.2.1Secured Communication Architecture . . . 133
B.3.1Reliable Communication Architecture . . . 151
B.4.1Rule: Security Introduction . . . 164
B.4.2Rule: Reliability Introduction . . . 165
B.5.1A Bigger Architecture . . . 165
B.5.2Secured Communication Architecture . . . 166
B.5.3Aspect Intro: Reliable Communication Architecture . . . 167
B.5.4Reliable-Secured Communication . . . 168
B.5.5Secured-Reliable Communication . . . 169

List of Tables

1 An Example of a General Conformance Check 73

1 Introduction, Related Work

Sections 1.1-1.4 are a guided tour that explains the motivation behind this research. Sections 1.5 and 1.6 support the motivation by concisely exposing the research question and its solution. Section 1.7 introduces aspects, aspect-oriented programming, AOP terminology, explains the benefits of AOP and exposes its limitations. Weaving is a special type of transformation. Some of the model weaving work conducted by the modeling community and its relation with our research is explained in section 1.8.

1.1 Motivation

The requirements of a system are the description of the system in terms of the services that it provides, and the constraints applied on its operation (Sommerville, 2011; Radatz et al., 1990). Modularization is a mechanism for decomposing a system into component parts based on some principle: functional decomposition, architectural decomposition and so on. In functional decomposition, each component is an independent work assignment unit that does one thing and one thing only. Modularization techniques facilitate design and development by improving the flexibility and comprehensibility of a system along with shortening its development time (Radatz et al., 1990; Parnas, 1972; Sommerville, 2011). The relationship between the requirements and the components (decomposition of a system) is not always straightforward. Regardless of the size of the system, a single requirement might not be implemented by a single component and some components might implement more than one requirement (Sommerville, 2011). The decomposition and composition supported by a given formalism together with system requirements dictate the choice of boundaries for separation of concerns (Tarr et al., 1999).

1.2 Modularization via "Aspects"

In spite of the wide acceptance of separation of concerns as a good software engineering principle, it is hard to find a useful definition of "concern," because most attempts to define concerns try to relate them to programs . "In fact, as discussed by Jacobsen and Ng (2004), concerns are really reflections of the system requirements and priorities of stakeholders in the system" (Sommerville, 2011). Depending on the stakeholder, concerns can be classified into different types: functional concerns, quality of service concerns, policy concerns, system concerns, organizational concerns and so on. All these concerns can be classified mainly into two groups, i) core concerns, and ii) system/cross-cutting concerns. Cross-cutting concerns are distinguished from the core concerns because they cross-cut through different core concerns. They are also called system concerns because they apply to the whole system rather than to individual components (Sommerville, 2011, in particular Chapter 21). If we relate concerns with requirements, then aspects are cross-cutting concerns or system concerns that cross-

cut through different core concerns or apply to a whole system (Kiczales et al., 1997). Some good examples of aspects are performance, reliability, security, authorization, synchronization, error handling (Kiczales and Mezini, 2005; Sommerville, 2011).

Although object-oriented, procedural and functional programming languages (Kiczales et al. (1997) refer to these languages as a "generalized-procedure" programming language) are well accepted by the entire software engineering community for abstraction, modularization and have many other significant advantages, but none of them can clearly address the problem of aspects/cross-cutting concerns in actual code (Elrad et al., 2001b; Kiczales et al., 1997; Laukkanen, 2008). The implementation of cross-cutting concerns by generalized-procedure languages will produce code where aspects will be spread out over different modules and the final code will be tangled, and challenging to develop, combine and maintain (Kiczales et al., 1997; Lieberherr et al., 2001).

1.3 AOP to the Rescue?

Aspect-oriented programming (Kiczales et al., 1997; Laukkanen, 2008) is a complementary programming technique to the generalized-procedure programming languages. 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) (Kiczales et al., 1997; Laukkanen, 2008; Díaz Pace and Campo, 2001; Elrad et al., 2001a).

Though the aspect-oriented community claims that AOP has some useful characteristics, unfortunately, it has some significant drawbacks as well. For the very same reason for which Dijkstra (1968) in his famous letter considered goto statements harmful, Constantinides et al. (2004) characterize AOP as the modern-day "OOP goto" and even provocatively asked if AOP is 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 (Laukkanen, 2008).

Moreover, AOP is implementation-dependent and too concrete. The abstraction, information hiding, modularity, and compositional reasoning in AOP conflict with the traditional way these principles are viewed (Kiczales and Mezini, 2005). The developers find implementations hard to understand, add to the system, and maintain. However, not only the methodology, but also having to deal with aspects at the programming language level is one of the reasons that leads AOP to many of these problems.

1.4 Raise the Level of Aspect Introduction

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. Software architecture (Commission et al., 2011; Garlan and Shaw, 1994; Len et al., 2003), is a comprehensible higher level abstraction of an overall system structure influenced by different stakeholders. It has powerful features that reflect various aspects of the software system, i.e., system understanding, analysis and evaluation, construction, evolution, reuse, stakeholders' communication, team organization, and hence, can play a vital role in software development (Qin et al., 2010; Garlan, 2000). The role of software architecture in software development does not concentrate on any single stage of the software development life cycle, but depending on the degree of use it is noticeable in almost every phase of the software development life cycle.

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. Introducing aspects at the architecture level allows 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 would be more comprehensive, and design decisions will be clearly captured in the actual code. It also influences some other benefits regarding 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 exist due to feature interactions.

1.5 Research Question

In this thesis, 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 (2.2.2) representing the architecture and applying the aspect by performing the graph transformation. To illustrate 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?

1.6 Summary of Contributions

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 ar-

chitecture or system specification. Fiadeiro and Maibaum (1992, 1990) 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 principles of data abstraction, scope, and encapsulation. To combine the components, the category-theoretic notion of colimit is used. The technique used to generate the system specification from components and connectors is independent of the underlying logic and successfully models software engineering principles such as modularity, inheritance, incrementality, reusability, and other related concepts (Fiadeiro and Maibaum, 1992, 1996, 1995). 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 (see section 3.2.1.4), in order to perform the graph transformation, we introduce a new transformation technique. The technique is inspired by the hyperedge replacement graph transformation technique. Besides that, the category-theoretic notions of kleisli category, monad and pushout also make some contribution to defining the transformation, and help us to figure out a structured way to introduce an aspect at the architecture level by performing a transformation.

In addition to proposing the transformation technique, we have implemented tool support based on Hets, an initiative of Mossakowski et al. (2007). Although our approach to aspect introduction via diagram transformation is in principle independent of the underlying specification category, we have not found it feasibe to create plausible examples in the languages currently supported by Hets, e.g., CASL, CoCASL, OWL. Therefore, the core of our tool support is the implementation of the logic of Fiadeiro and Maibaum (1992), under the name MFLogic. The remaining contribution of the tool implementation consists of support for automatic aspect introduction and analysis.

1.7 Related Work: Aspect Oriented Programming

To manage the complexity of software systems, different kinds of programming paradigms have been developed, and aspect-oriented programming is one of the results of this evolution. The abstract representation of the requirements and the relationship between the representation and its implementation is not always straightforward. Despite having a rich set of tools, the modular implementation of cross-cutting concerns by the generalized-procedure programming paradigm (i.e., procedural, OO, Functional) will produce "tangled" code. Aspect-oriented programming is an auxiliary programming technique to the generalized-procedure programming paradigm that provides a means

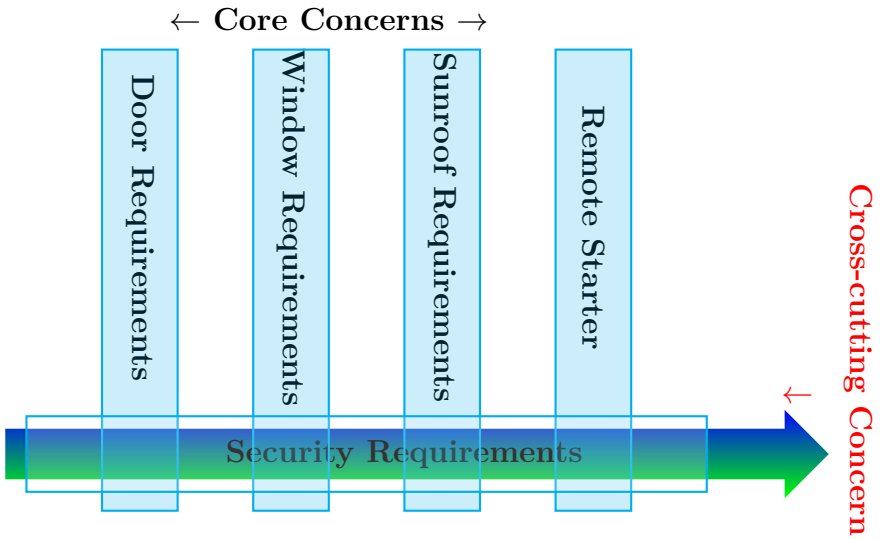


Figure 1.7.1: Aspect: security requirements

to address the well-known problem of separation of cross-cutting concerns. In aspect-oriented programming, the total system implementation is divided into two main parts: the *base program* (i.e., object-oriented program) and the *aspect program*. The base program will reflect the core, i.e., functional, requirements of the system and can be implemented by using any generalized-procedure programming paradigm. On the other hand, the aspect program will represent the cross-cutting functionality of the system and would be implemented by the aspect-oriented program. Finally, the well-defined modular implementation (i.e., "aspect program") of cross-cutting concerns (i.e., aspects) will lead to a more structured system development.

For a clearer understanding of the concept of aspect-oriented programming, one needs to understand the common terminology associated with it. The description of the key terminology associated to AOP are summarized as follows:

Aspect/Cross-cutting Concern: If we think of concerns as a way of organizing the requirements and priorities of stakeholders, then *aspects*, also known as cross-cutting concerns, are the system concerns that cross-cut through different main (also called functional/core) concerns. An example of an aspect/cross-cutting concern is depicted in Figure 1.7.1.

Advice: This is the code (i.e., method, class) implementing a cross-cutting concern as an individual component. This implementation is neither concerned with how this component is related to others nor with how other components will be implemented. According to Figure 1.7.1, advice would be the implementation of the security requirements.

Join Point: *Join points* are the events in an executing program (i.e., all places in a code) where advice may be executed. Some examples of *join points* are method and constructor calls, exception handlers and advice executions or modifications of a

class's attributes.

Pointcut: A *pointcut* is a statement that defines the *join points* for a relevant aspect. Put in another way, we can say that the *pointcut* is a set of join points (i.e., composed like predicates using &&, ||, !) where the associated aspect advice should be executed.

Weaving: The process of incorporating advice code to its associated join points is called *weaving*. There are different types of weaving, namely: compile-time, load-time or runtime weaving. A separate compiler handles compile-time weaving. The load-time weaving utilizes the classloader to perform the weaving. Proxy classes and code generation libraries are responsible for runtime weaving.

Aspect Program: An *aspect program* is a modular implementation of an aspect or cross-cutting concern. It includes a set of *pointcuts* and related *advice*.

In the following section, by illustrating the above terminologies associated with AOP, we illustrate the concept of aspect-oriented programming. In order to demonstrate these, we have used an example (i.e., JHotDraw) introduced by Kiczales (2006) which is well-known in the AOP community.

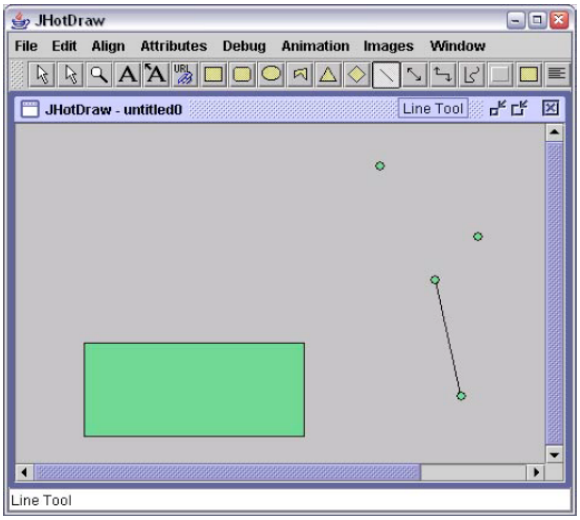


Figure 1.7.2: A Simple Drawing Tool (Kiczales, 2006)

JHotDraw (Figure 1.7.2) is a simple drawing tool that enables drawing points, lines, rectangles and other shapes. This tool system has another property called the display update property. Whenever any shape (i.e., point, line) changes, the display should be updated. By keeping object (object-oriented development) in mind, if we want to do the design of the above requirement, we may come up with the UML Class diagram in Figure 1.7.3.
If we use object-oriented programming to implement the above design, one of the many implementations would look like Figure 1.7.4.

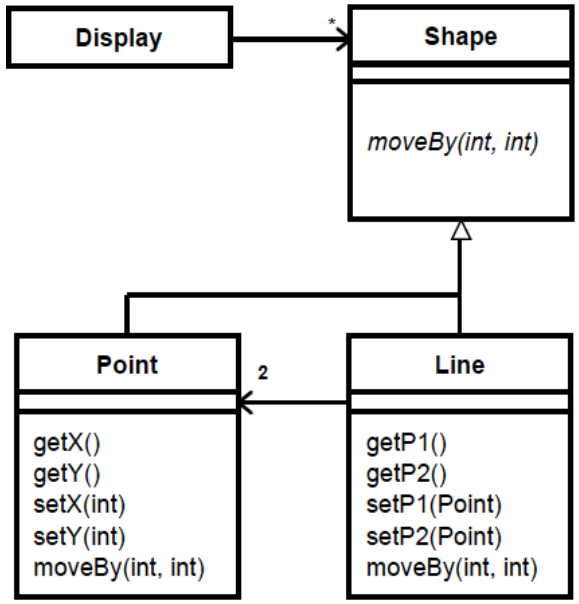


Figure 1.7.3: OO design (Kiczales, 2006)

```
class Point extends Shape {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void moveBy(int dx, int dy) {
    x = x + dx; y = y + dy;
  }

  void setX(int x) {
    this.x = x;
  }

  void setY(int y) {
    this.y = y;

  }
}
```

Figure 1.7.4: OO program (Kiczales, 2006)

If we want to observe the display update property, we will have to trace and monitor

the changes in different places. The design in Figure 1.7.5 shows the places we will have to monitor.

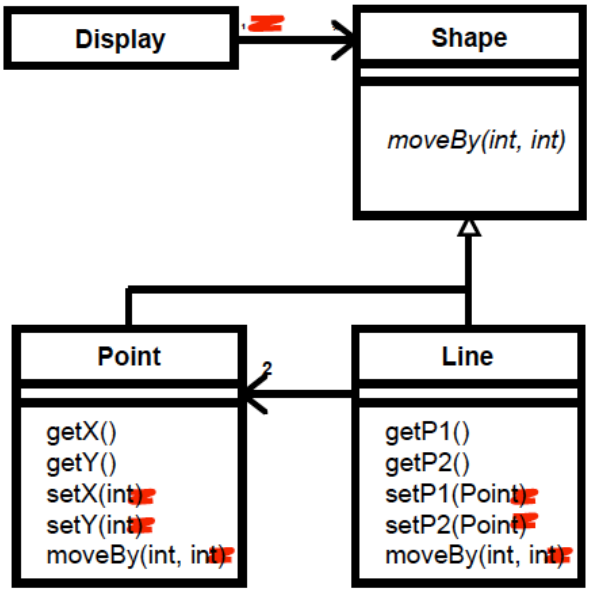


Figure 1.7.5: OO design with aspect (Kiczales, 2006)

Our new implementation would look like the Figure 1.7.6, where the display update property is scattered into different methods, and the implementation is no longer modular.

```
class Point extends Shape {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void moveBy(int dx, int dy) {
    x = x + dx; y = y + dy;
    display.update(this);
  }
  void setX(int x) {
    this.x = x;
    display.update(this);
  }
  void setY(int y) {
    this.y = y;
    display.update(this);
  }
}
```

Figure 1.7.6: OO Scattered Code (Kiczales, 2006)

If we want to resolve this problem and make the system development modular, the design of the system will reflect Figure 1.7.7.

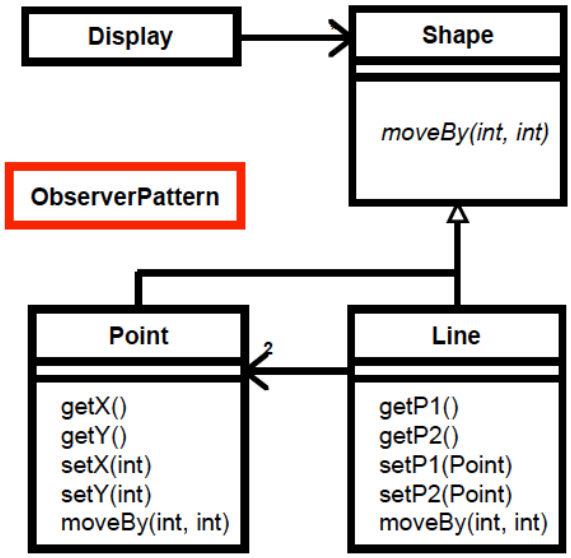


Figure 1.7.7: Aspect design (Kiczales, 2006)

When we go for implementation, the OO program will remain as it was, and the display update property will be implemented separately as an aspect program (i.e., *ObserverPattern*). The aspect program will look like Figure 1.7.8:

```
aspect ObserverPattern {

  pointcut change():
    execution(void Shape.moveBy(int, int)) ||
    execution(void Line.setP1(Point))      ||
    execution(void Line.setP2(Point))      ||
    execution(void Point.setX(int))        ||
    execution(void Point.setY(int));

  after() returning: change() {
    Display.update();
  }
}
```

Figure 1.7.8: Aspect program (Kiczales, 2006)

Here ObserverPattern is the aspect, and it consists of advice and a pointcut. **Display.update()** is the advice and **change()** is the pointcut. The pointcut defines and consists of five join points. Now the system implementation is modular because the cross-cutting concern "display update" is moved away from the base program to a separate module ObservePattern.

But aspect-oriented programming has many drawbacks. Some of the major drawbacks are that its modular structure is implicit, it is implementation dependent, and

it can lead to some problems that would be untraceable.

While we are defining pointcut, we may implant the untraceable problem by selecting wrong join points. Consider Figure 1.7.9 as an example to illustrate the possibilities of defining the wrong pointcut.

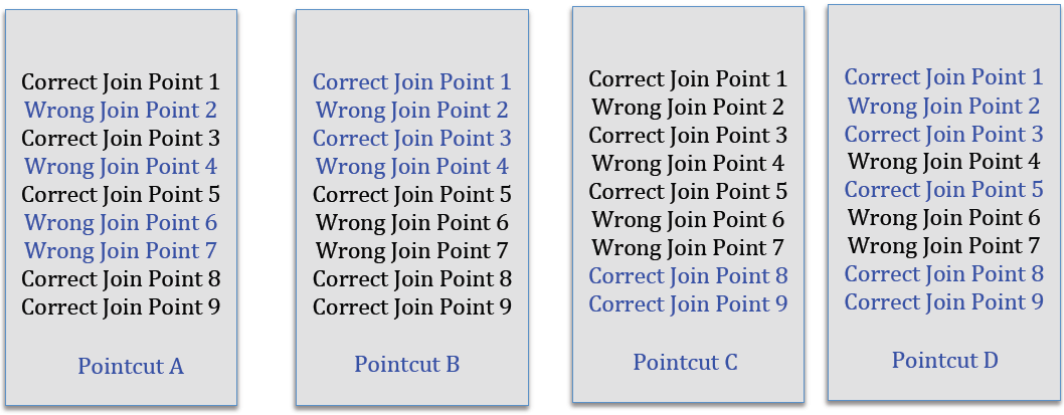


Figure 1.7.9: Possible error situations with pointcut

Suppose we have nine join points in a class, but five of them are associated with a particular aspect advice. So, according to our above example, the correct pointcut is the set of five join points (e.g., correct join points are 1,3,5,8,9). But as we can see, there are at least four natural ways one can make an error and lead to intractability problems (Lemos et al., 2006). Here the blue color join points are the members of the defined pointcut (e.g., Pointcut A={Join Point 2,4,6,7}).

A Pointcut selects none of the associated but all unassociated join points.

B Pointcut selects some associated and some unassociated join points.

C Pointcut selects some associated join points but not all.

D Pointcut selects all associated but some unassociated join points.

1.8 Related Work: Model Transformation

The aspect-oriented software development (AOSD) paradigm first appeared at the University of Twente in the Netherlands at the code level. Different types of aspect-oriented programming paradigm have been since developed (e.g., subject-oriented programming, feature-oriented programming, adaptive programming), and the most popular among these is the "aspect-oriented programming".

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 through ad-hoc (model and meta model specific) or generic aspect-oriented modeling (AOM) approaches. Some of the generic AOM approaches are: MATA by Jayaraman et al. (2007), and by Whittle and Jayaraman (2007), Generic SmartAdapter by Morin et al. (2007), and by Morin et al. (2008a) and Kompose by France et al. (2007), and by Fleurey et al. (2007). Most of the modeling aspects work is done by using UML, the Unified Modeling Language. Modeling aspects in UML can be classified into three broad categories. The first approach is aspect composition by using a generic merge algorithm, and it includes the work done by Siobhan and Elisa (2005), and by France et al. (2004). The second approach is weaving aspects by putting an emphasis on the principle of reuse (AOP mechanism at the modeling level), and examples of this type include the work accomplished by Cottenier et al. (2007), Jacobson and Ng (2004). The third approach is weaving aspects by applying existing graph transformation techniques; two examples of this type are the works conducted by Whittle and Jayaraman (2008), and by Morin et al. (2008b).

These three approaches are not sufficient for aspect introduction or weaving. Whittle and Jayaraman (2008) explained, in detail, the limitations of the first and second approaches. In brief, the first approach is not expressive enough to handle all model compositions and the second approach is overly restrictive. The third approach tackles all of the problems related to the first and second approaches, but the limitation of this third approach is that it can not cover all (or even some general) scenarios that evolve after introducing aspects in software architectures. Since model weaving is a special case of transformation; some interesting work in weaving aspects into models are explained in the following sections along with their similarities and dissimilarities with our work.

Whittle and Jayaraman (2008) 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 MATA, models are an instance of a type graph and the composition of an aspect model with a base model is specified by a graph transformation rule $r : LHS \rightarrow RHS$; defined over the typed graph. In a rule r, the LHS (left-hand side) defines the pointcuts (where to add elements), and the RHS (right-hand side) defines the advice. Hence, a rule in MATA identifies the places where new aspect elements need to be added, what elements need to be added, and how they should be added. 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), (this is similar to the approach applied in VIATRA developed by Csertán et al. (2002)) which allowed them to write a rule on a single model instead of repeating unchanged elements in both the LHS and the RHS. Figure 1.8.1 explains, (a) a MATA graph rule and (b) the application of the rule to a particular sequence diagram. The rule $R1$ advices

creation of two messages r, and s whenever it finds a matching of the message p in a base sequence diagram. The message p would remain unchanged.

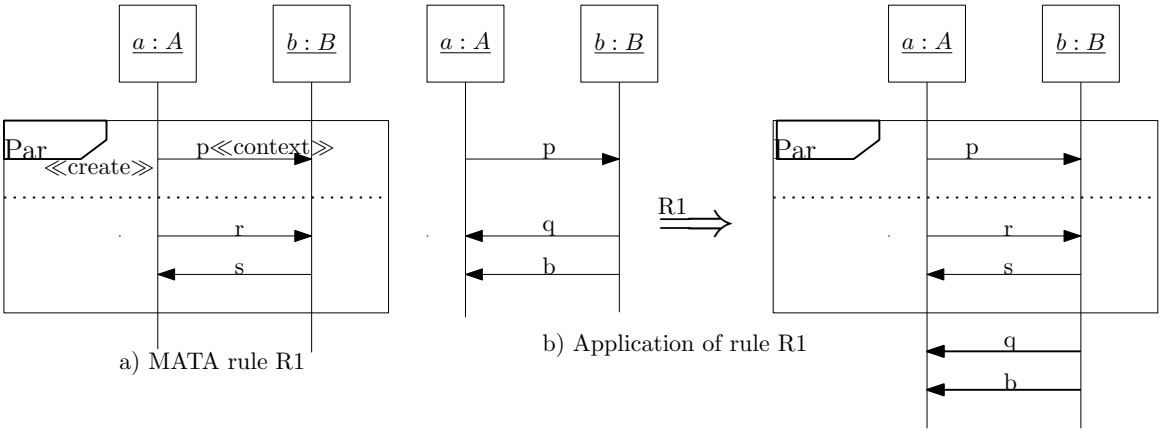


Figure 1.8.1: MATA rule and its application

Morin et al. (2008b) 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 model is called the pointcut, and the two morphisms are defined from the pointcut to the base and to the advice, respectively. The morphisms prescribe the deletion, preservation, and addition. This weaving process is similar to defining a rule in the double-pushout graph transformation approach explained in section 2.4.1.

Our work introduces aspects at the earlier phase of the software development lifecycle, i.e., at the architecture level, by performing graph transformation. Although, regarding goals, we have some similarity with the work of the AOM community, our approach is completely different from their approaches. The way Kienzle et al. (2009); Morin et al. (2008b) explained aspects contradicts their traditional definition. We diverge from AOM weaving by 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 are composed to produce a weaved model. In contrast, our 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 pre-defined rules. Though neither the rules nor the resultant architecture is wrong.

As an example, consider the following Figure 1.8.2. It contains, a) the MATA rule R2, b) the MATA rule R3, c) the application of rule R2, and d) the application of rule R3. The rule R2 prescribes the deletion of message t, the creation of two parallel messages r and s, and keeps message p unchanged. The rule R3 prescribes the deletion of message t, keeps p unchanged and, after matching any number of messages, creates two parallel messages u and v. After successfully applying the rule R2, we get the result sequence diagram. Now, if anyone wants to apply the rule R3 on the new resulting sequence diagram, he/she will experience a conflict (unable to apply the rule) due to the nonexistence of the message t. But neither the rule R3 nor the result sequence diagram is wrong. Moreover, changing the sequence cannot resolve this conflict either.

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 *zigzag graph transformation* technique is capable of addressing this issue. Besides that, it allows us to automatically verify the conformance check (property preservations) of the new system architecture with the old system architecture.

1.9 Overview of the Thesis

This thesis proceeds as follows: chapter 2 contains the description and summary of some of the technologies and their terminologies associated with this research, i.e., graphs, category-theoretic notions, relational-algebraic notions, graph transformation essentials and present graph transformation techniques. In this chapter, we also describe the logic, i.e., the MFLogic we are using to describe the components, connectors, and the system architecture. Chapter 3 illustrates our research challenges, analyzes the application of the established graph transformation techniques in our settings, addresses our transformation notations and methodology by precisely representing our methodological terms and methodologies (formal definition). Then, in chapter 4 we analyze the effect of aspect introduction, prove the partial pushout existence and some other theorems. Chapter 5 contains meta theorems and shows that proof of property preservation (conformance check of the new system architecture with the old architecture) is automatic. An overview of the tool implementation and how to use it is explained in chapter 6. The conclusions followed by some potential future work are mentioned in chapter 7. Finally, in appendix B, with a well-known example of the sender-receiver communication, we illustrate the research goal.

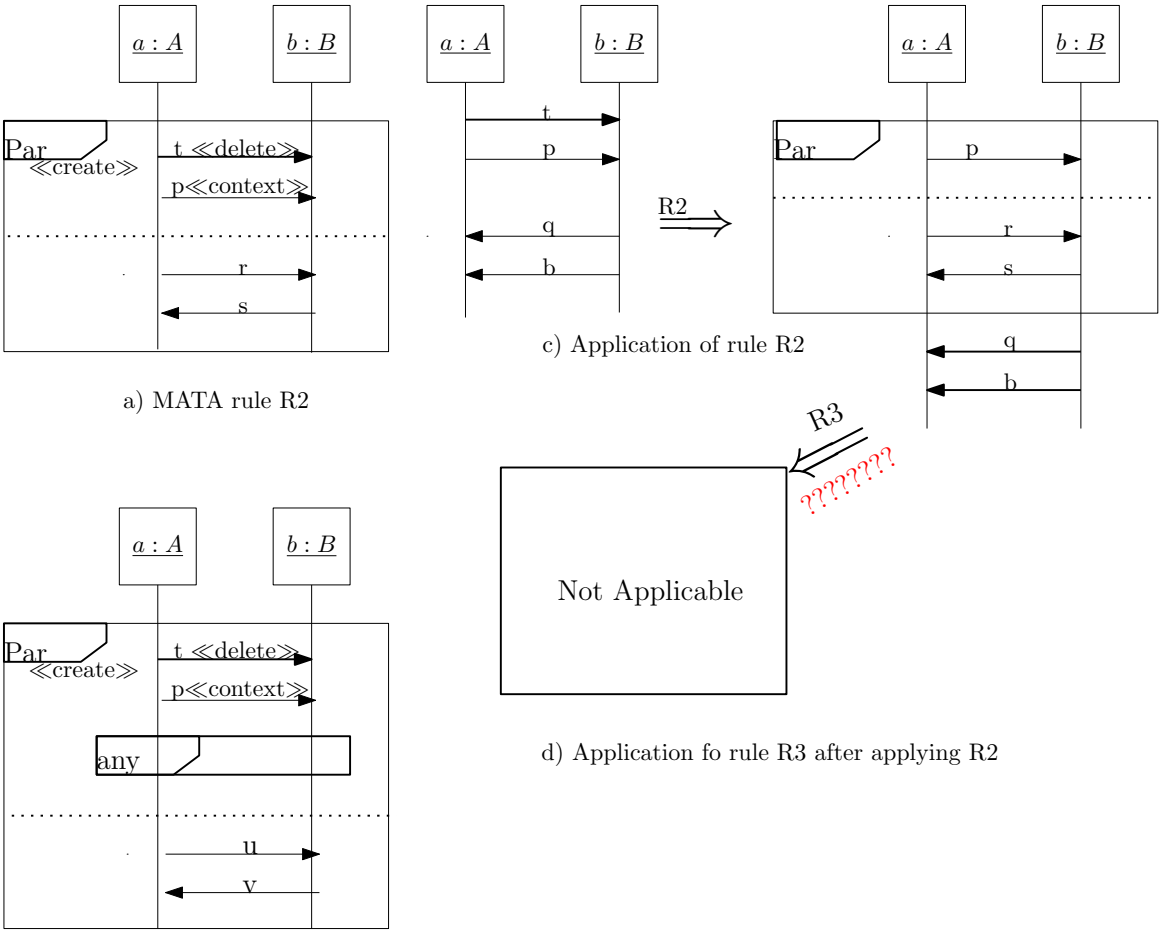


Figure 1.8.2: MATA rule and its limitation

2 Background

This chapter begins with the definition of a graph and proceeds with the description of category-theoretic and relation-algebraic notations most relevant to this research. We summarize the graph transformation essentials and some established (i.e., double-pushout, single-pushout) graph transformation techniques in section 2.4. Finally, in section 2.5 we outline the logic introduced by Fiadeiro and Maibaum (1992) in their "temporal theories as modularisation units for concurrent system specification", and its technique we are using to describe the components, connectors and to construct the system specification from components and connectors.

2.1 Graphs

Our research leverages directed graphs (the edges will have direction). The formal definition of our graphs is as follows:

Definition 2.1.1. ***(Graph)*** *A graph $\mathscr{G}$ is a tuple $\mathscr{G} := (\mathscr{V}, \mathscr{E}, src, tgt)$ where*

- *$\mathscr{V}$ is the finite set of vertices of $\mathscr{G}$*
- *$\mathscr{E}$ is the finite set of edges/arrows of $\mathscr{G}$*
- *$src : \mathscr{E} \to \mathscr{V}$ is the source function for $\mathscr{G}$ and*
- *$tgt : \mathscr{E} \to \mathscr{V}$ is the target function for $\mathscr{G}$.*

2.2 Category-theoretic Notions

This section contains the basic category-theoretic notations most relevant to our research. For further details, please follow Barr and Wells (1990) and Simmons (2011).

Definition 2.2.1. ***(Category)*** *A category $\mathscr{C}$ is a tuple $\mathscr{C} := (Obj_{\mathscr{C}}, Mor_{\mathscr{C}}, src, tgt, id, \circ)$ where*

- *$Obj_{\mathscr{C}}$ is a collection of entities called objects*
- *$Mor_{\mathscr{C}}$ is a collection of entities called morphisms (also called arrows). $Hom_{\mathscr{C}}(A, B)$ represents collection of all morphism (may be zero or many) from A to B, where $A, B \in Obj_{\mathscr{C}}$.*
- *src, and trg are two assignments $src, trg : Mor_{\mathscr{C}} \to Obj_{\mathscr{C}}$. For each arrow f where $f \in Hom_{\mathscr{C}}(A,B)$, we write $f : A \to B$ to indicate that $A = src(f)$ and $B = trg(f)$.*

- *id is an assignment for every object $A \in Obj_{\mathscr{C}}$, ie., $id : Obj_{\mathscr{C}} \rightarrow Hom_{\mathscr{C}}(A,A)$ called identity morphism.*
- *$\circ$ — a composition operator, $\circ : Hom_{\mathscr{C}}(A,B) \times Hom_{\mathscr{C}}(B,C) \rightarrow Hom_{\mathscr{C}}(A,C)$. Composition of two morphisms $f : A \rightarrow B$ and $g : B \rightarrow C$ is $g \circ f : A \rightarrow C$.*

such that the following hold:

- *Associativity — any morphisms $f \in Hom_{\mathscr{C}}(A, B)$, $g \in Hom_{\mathscr{C}}(B, C)$, and $h \in Hom_{\mathscr{C}}(C, D)$ satisfy $h \circ (g \circ f) = (h \circ g) \circ f$;*
- *Identity — any morphisms $f \in Hom_{\mathscr{C}}(A, B)$, and $g \in Hom_{\mathscr{C}}(B, A)$ satisfy: $id_B \circ f = f$ and $g \circ id_B = g$*

It is clear that every category $\mathscr{C}$ (finite) has an underlying graph which will be denoted by $|\mathscr{C}|$. The nodes, edges, source and target maps of $|\mathscr{C}|$ are objects, arrows, source and target maps of $\mathscr{C}$. A homomorphism $\mathscr{H} : \mathscr{G}_1 \rightarrow \mathscr{G}_2$ between two graphs $\mathscr{G}_1$ and $\mathscr{G}_2$ is a function taking nodes to nodes and edges to edges and preserving the source and the target maps.

Definition 2.2.2. ***(Diagram)*** *A* diagram *in a category $\mathscr{C}$ is a graph homomorphism $\mathscr{D} : \mathscr{I} \rightarrow |\mathscr{C}|$ for some graph $\mathscr{I}$. The graph $\mathscr{I}$ is called the* index graph *or the* shape graph *of the diagram. Such a diagram is called a diagram of type $\mathscr{I}$ (Barr and Wells, 2013, 1990).*

A functor F between two categories is a graph homomorphism which preserves identities and composition. Functors are useful for transformations from one type of mathematics to another.

Definition 2.2.3. ***(Functor)*** *Given a pair of categories, a* functor *$F : \mathscr{C} \rightarrow \mathscr{D}$ consists of:*

- *Object assignment — $F_{Obj} : Obj_{\mathscr{C}} \rightarrow Obj_{\mathscr{D}}$;*
- *Morphism assignment — $F_{Mor} : Mor_{\mathscr{C}} \rightarrow Mor_{\mathscr{D}}$, e.g., $f : A \rightarrow B$ in $\mathscr{C}$, $F_{Mor}(f) : F_{Obj}(A) \rightarrow F_{Obj}(B)$ in $\mathscr{D}$.*

such that:

1. *F preserves identity arrows: For an object A, the identity on A, namely $id_A : A \rightarrow A$, is mapping to the identity on the images of A, namely $id_{F_{Obj}(A)} : F_{Obj}(A) \rightarrow F_{Obj}(A)$, that is, $F_{Mor}(id_A) = id_{F_{Obj}(A)}$*
2. *F preserves composition: $F_{Mor}(g \circ f) = F_{Mor}(g) \circ F_{Mor}(f)$ for all morphisms $f : A \rightarrow B$ and $g : B \rightarrow C$ in $\mathscr{C}$.*

Definition 2.2.4. ***(Cocone)*** *(Barr and Wells, 2013, 1990) Let $\mathscr{G}$ be a graph and $\mathscr{C}$ be a category and let $D\colon \mathscr{G} \to |\mathscr{C}|$ be a diagram in $\mathscr{C}$ with shape graph $\mathscr{G}$. A* cocone *with base D is an object C of $\mathscr{C}$ together with a family $(u_a)_{a \in \mathscr{V}_\mathscr{G}}$ of arrows of $\mathscr{C}$ indexed by the nodes of $\mathscr{G}$, such that $u_a : D(a) \to C$, for each node a of $\mathscr{G}$.*

The cocone is *commuting* if for each arrow s : a → b of $\mathscr{G}$, the diagram in Figure 2.2.1 commutes, i.e., $u_b \circ D(s) = u_a$.

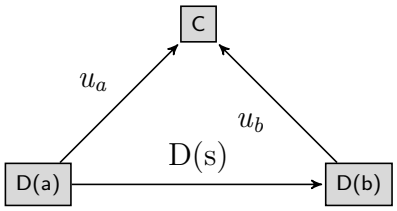


Figure 2.2.1: Commuting cocone

Definition 2.2.5. ***(Colimit)*** *(Barr and Wells, 2013, 1990) A commuting cocone over the diagram D is called universal if it has a unique arrow to every other commuting cocone over the same diagram. A universal cocone, if such exists, is called a* colimit *of the diagram D.*

For reference, we spell out the special case of pushouts:

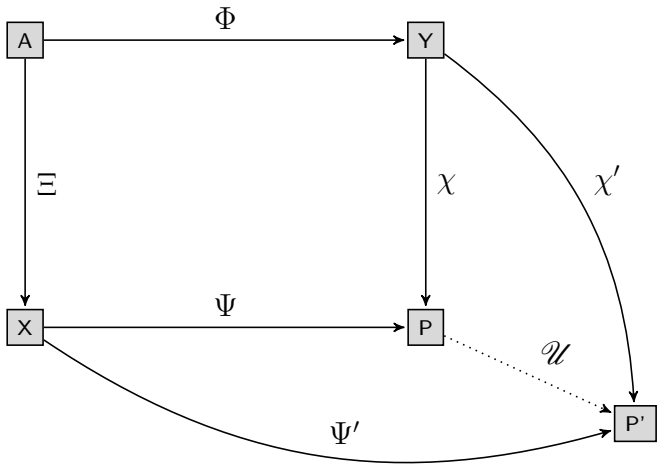
Definition 2.2.6. ***(Pushout)*** *For given a span $X \xleftarrow{\Xi} A \xrightarrow{\Phi} Y$ (three objects and two arrows) in a category $\mathscr{C}$, the* pushout *is a cospan $X \xrightarrow{\Psi} P \xleftarrow{\chi} Y$ (one new object and two arrows), such that:*

1. *(commuting Main Square) — $\chi \circ \Phi = \Psi \circ \Xi$*
2. *(Cospan Universal) — Given any outer commuting square, i.e., given $\chi' : Y \to P'$, $\Psi' : X \to P'$ for which $\chi' \circ \Phi = \Psi' \circ \Xi$, there is:*
   - *A unique arrow $\mathscr{U} : P \to P'$ such that:*
     - *$\mathscr{U} \circ \Psi = \Psi'$ and*
     - *$\mathscr{U} \circ \chi = \chi'$*

Pushouts for spans equivalently expand to colimits by adding the induced morphism from A to P.

2.3 Relation-algebraic Notations and Properties

The zigzag paths that will be defined in section 3.3 consist of directed edges traversed in an arbitrary directions. Functions and function composition are not flexible



enough to represent any association between the structures associated with *source* and *target* nodes of a zigzag path in a diagram. On the other hand, relations will be flexible, and allow to draw a relation between the *source* and *target* of a zigzag path. For some proofs in section 3.7 and section 4.3 we will need relational versions of homomorphisms, and therefore now introduce some basic relation-algebraic notations and concepts. For further details, please follow Schmidt and Ströhlein (1993), Kahl (2002), and Kahl (2004).

- A relation $\mathscr{H}_v$ between two sets $\mathscr{V}_1$ and $\mathscr{V}_2$ is declares as $\mathscr{H}_v : \mathscr{V}_1 \leftrightarrow \mathscr{V}_2$.
- Identity relations are represented as $\mathbb{I}$
- The composition is defined as forward composition. For given two relation $\mathscr{H}_{v1} : \mathscr{V}_1 \leftrightarrow \mathscr{V}_2$ and $\mathscr{H}_{v2} : \mathscr{V}_2 \leftrightarrow \mathscr{V}_3$, the composition is: $\mathscr{H}_{v1} ; \mathscr{H}_{v2} : \mathscr{V}_1 \leftrightarrow \mathscr{V}_3$.
- The *inclusion* is represented as $\sqsubseteq$. $\mathscr{H}_{v1} \sqsubseteq \mathscr{H}_{v2}$ means $v_1, v_2 : ((v_1, v_2) \in \mathscr{H}_{v1} \Rightarrow (v_1, v_2) \in \mathscr{H}_{v2})$
- The *transpose* or *converse* of a relation $\mathscr{H}_{v1} : \mathscr{V}_1 \leftrightarrow \mathscr{V}_2$ is ${\mathscr{H}_{v1}}^{\smile} : \mathscr{V}_2 \leftrightarrow \mathscr{V}_1$; it is defined as ${\mathscr{H}_{v1}}^{\smile} = \{(v1, v2) | (v2, v1) \in \mathscr{H}_{v1}\}$
- A relation $\mathscr{H}_{v1} : \mathscr{V}_1 \leftrightarrow \mathscr{V}_2$ is called *total* iff every element $v_1 \in \mathscr{V}_1$ has at least one image $v_2 \in \mathscr{V}_2$; that is, $\mathbb{I} \sqsubseteq \mathscr{H}_{v1} ; {\mathscr{H}_{v1}}^{\smile}$
- A relation $\mathscr{H}_{v1} : \mathscr{V}_1 \leftrightarrow \mathscr{V}_2$ is called *univalent* say element $v_1 \in \mathscr{V}_1$ maps to at most one element $v_2 \in \mathscr{V}_2$, that is, ${\mathscr{H}_{v1}}^{\smile} ; \mathscr{H}_{v1} \sqsubseteq \mathbb{I}$.
- A *function* is a univalent and total relation.

Definition 2.3.1. ***(Relational graph homomorphism)*** *For two given graphs* $\mathscr{G}_i = (\mathscr{V}_i, \mathscr{E}_i, src_i, tgt_i)$ *for* $i \in \{1, 2\}$, *a* relational graph homomorphism $\mathscr{H} : \mathscr{G}_1 \leftrightarrow \mathscr{G}_2$

consists of two relations $\mathscr{H}_v : \mathscr{V}_1 \leftrightarrow \mathscr{V}_2$ *and* $\mathscr{H}_e : \mathscr{E}_1 \leftrightarrow \mathscr{E}_2$ *that satisfy the inclusion* $\mathscr{H}_e{}^{\smile} ; src_1 \sqsubseteq src_2 ; \mathscr{H}_v{}^{\smile}$ *and* $\mathscr{H}_e{}^{\smile} ; tgt_1 \sqsubseteq tgt_2 ; \mathscr{H}_v{}^{\smile}$ *represented by the sub-commuting diagrams in Figure 2.3.1.*

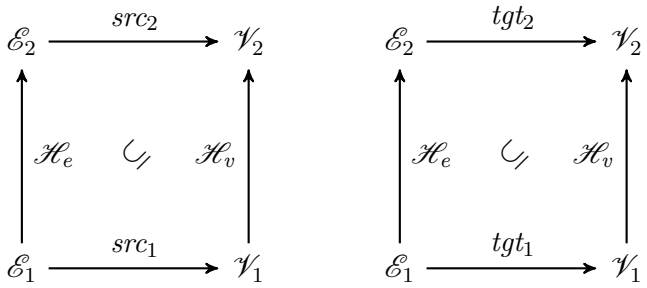


Figure 2.3.1: Relational graph homomorphism

A *graph homomorphism* is a univalent and total *relational graph homomorphism.*

Definition 2.3.2. ***(Category** RelGraphs)* *Graphs and their relation graph homomorphism form a category RelGraphs. It is a Dedekind category and according to the definition it has converse* $^{\smile}$.

Definition 2.3.3. ***(Functor** GraphsToRelGraphs)* *GraphsToRelGraphs is a functor between Graphs and RelGraphs category, denoted by* $\mathsf{F}_{\mathsf{GRG}}$.

2.4 Graph Transformation

Literally, a transformation of something means a change in its shape or appearance. In graph transformation, the underlying subject whose form is changed is a graph, and this change is controlled by some guiding principles called rules (often called production rules). Different kinds of visual notations have been used to describe several computer science processes (Ghezzi et al., 2002), and almost all of the visual techniques can be modeled using graphs. So, sophisticated graph transformation techniques are present directly or indirectly in almost every process of software engineering (Rensink, 2005; Heckel, 2006).

An introduction to the essential components, steps and standard nomenclature (Kahl, 2002; Heckel, 2006) generally used to describe graph transformation techniques is briefly explained in the following section:

- For some given graph, a graph transformation technique usually consists of a set of graph rewriting rules where a rule $r = \mathcal{L} \to \mathcal{R}$ consists of a name of the rule "r", a left-hand side (pattern graph) $\mathcal{L}$, and a right-hand side (replacement graph) $\mathcal{R}$, respectively. Besides the constructive meaning (a hint how a matching instance of the left-hand side will be replaced by an instance of the right-hand

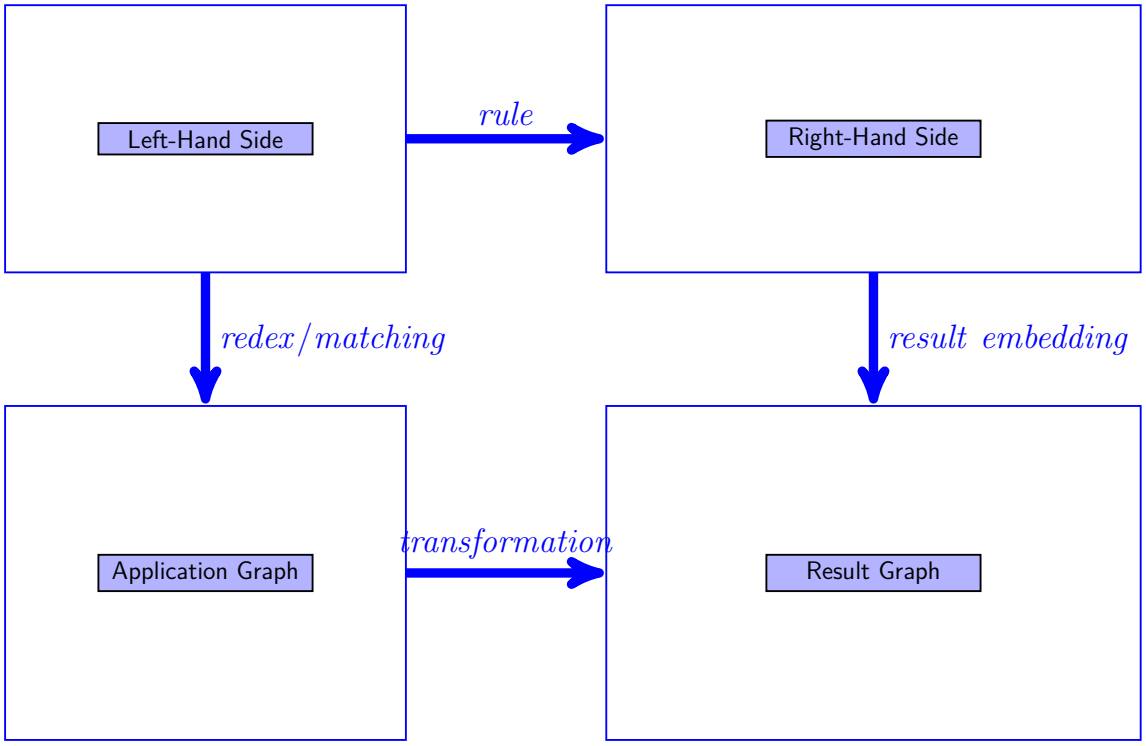


Figure 2.4.1: Introduction to Graph Transformation

side), rules generate a transformation by replacing an instance of the left-hand side by an instance of the right-hand side inside a given graph where the rule is applicable.

- 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 ($\mathcal{B}$). Generally we can get the result graph by deleting matched vertices and edges $\mathcal{L} - \mathcal{R}$ from $\mathcal{A}$ and pasting a copy of $\mathcal{R} - \mathcal{L}$ to the result. An instance of the right-hand side can be identified in the result graph through the result embedding.

2.4.1 Double Pushout

The double-pushout (DPO) graph transformation approach modeled by two pushout diagrams in the category *Graph* of graphs and total graph morphisms is the first of the algebraic approaches introduced by Ehrig et al. (1973). The application of a given

rule $r = \mathcal{L} \xleftarrow{\Phi_L} \mathcal{G} \xrightarrow{\Phi_R} \mathcal{R}$ (a span of total injective graph morphisms) to a graph $\mathcal{A}$ consists of the following steps:

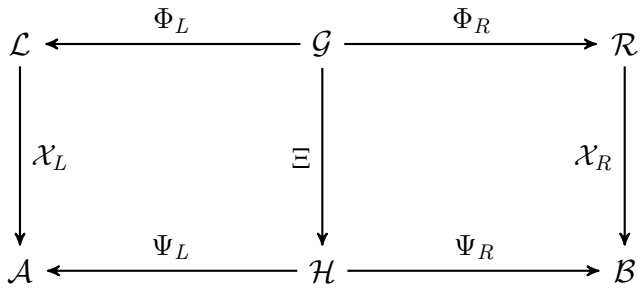


Figure 2.4.2: Double-pushout diagram

1. Need to find a graph morphism $\mathcal{X}_L : \mathcal{L} \to \mathcal{A}$ and check the following two conditions (gluing conditions):

   (a) Dangling condition: No edge in $\mathcal{A} - \mathcal{X}_L(\mathcal{L})$ is incident to a node in $\mathcal{X}_L(\mathcal{L}) - \mathcal{X}_L(\Phi_L(\mathcal{G}))$. For two graph morphisms $\Phi_L : \mathcal{G} \to \mathcal{L}$ and $\mathcal{X}_L : \mathcal{L} \to \mathcal{A}$ the dangling condition holds, iff whenever an edge connects a node outside the image of $\mathcal{X}_L$ with a node x inside the image of $\mathcal{X}_L$, then x has in its pre-image via $\mathcal{X}_L$ only nodes in the image of Φ_L.

   (b) Identification condition: For two graph morphisms $\Phi_L : \mathcal{G} \to \mathcal{L}$ and $\mathcal{X}_L : \mathcal{L} \to \mathcal{A}$ the identification condition holds iff whenever two discrete nodes of $\mathcal{L}$ are identified by $\mathcal{X}_L$, then they both lie inside the images of Φ_L. In other words, for all distinct items x,y $\in \mathcal{L}$, $\mathcal{X}_L$(x)= $\mathcal{X}_L$ (y) iff x,y $\in \Phi_L(\mathcal{G})$ (This condition is understood to hold separately for nodes and edges).

2. Pushout Complement: Find the host graph $\mathcal{H}$ and injective host graph morphism Ξ such that matching can be reconstructed from the psuhout of Φ_L and Ξ.

3. Construct the pushout in the category Graph of graphs and total morphisms $\Phi_R : \mathcal{G} \to \mathcal{R}$ and $\Xi : \mathcal{G} \to \mathcal{H}$ that will yield the result graph $\mathcal{B}$.

2.4.2 Single Pushout

The single-pushout (SPO) approach, introduced by Löwe (1993), is the second of the algebraic approaches and is modeled by a single pushout, a well-known universal construction in category theory, in the category *Graph* of graphs and partial graph morphisms. The application of a given rule $r = \mathcal{L} \xrightarrow{\Phi} \mathcal{R}$ (partial morphism) to a graph $\mathcal{A}$ consists of the following steps:

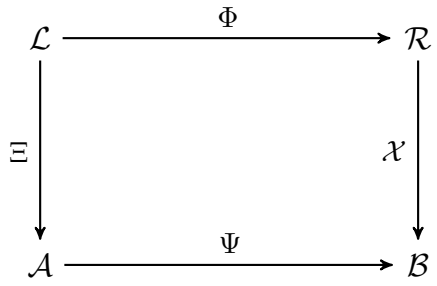


Figure 2.4.3: Single-pushout diagram

1. Choose a total graph morphism $\Xi : \mathcal{L} \to \mathcal{A}$
2. Construct the pushout in the category *PGraph* of graphs and *partial* morphisms that will yield the result graph $\mathcal{B}$. The pushout is constructed for the partial morphism $\Phi : \mathcal{L} \to \mathcal{R}$ and total morphism $\Xi : \mathcal{L} \to \mathcal{A}$.

A total matching morphism $\Xi : \mathcal{L} \to \mathcal{A}$ is called *conflict-free* for a (partial) rule morphism $\Phi : \mathcal{L} \to \mathcal{R}$ iff x $\in$ dom(Φ) and $\Xi(x) = \Xi(y)$, then, y $\in$ dom(Φ) or pre-image of Φ. A conflict-free matching $\Xi : \mathcal{L} \to \mathcal{A}$ ensures that the morphism $\mathcal{X} : \mathcal{R} \to \mathcal{B}$ is total. Partialities in the rule morphism may be interpreted as "prescription to delete", while defined parts of the rule morphism may be interpreted as "prescription to preserve".

2.4.3 Association between Graph Transformation and AOP

Graph transformation is a powerful tool, and it can be used to represent all the common terminologies associated with aspect oriented programming. The term aspect has the same meaning for both AOP and graph transformation technique. 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" in application graph is used to represent the term pointcut. The way a transformation is generated is equivalent to weaving and the combination of both the base and aspect programs is represented by the "result graph".

2.5 Introduction to MFLogic

This section presents the formal representation of the logic and the technique we are applying to specify the components, connectors and to construct the system specification from the components and connectors respectively. Section 2.5.1 defines the logic we are using to specify the individual components and connectors of an architecture. The logic was introduced by Fiadeiro and Maibaum (1992). The specification

of a component consists of a signature and a collection of axioms that represent the description of the component.

2.5.1 Component Signature and Specification

A component signature consists of three different elements: the *universe* element, the *attribute* element, and the *action* element. The state independent information is represented by the *universe* element. The *attribute* element includes the information that is state dependent. An *attribute* will change its state depending on the *action* applied to it.

Definition 2.5.1. ***(Component Signature)*** *A component signature is a triple* (Σ, A, Γ) *where*

- $\Sigma = (S, \Omega)$ *is a signature in the usual algebraic sense (Ehrig and Mahr, 1985), i.e., S is a set of sorts and Ω is a $S^* \times S$-indexed family of function symbols.*
- A *is a $S^* \times S$-indexed family of attribute symbols.*
- Γ *is an S^*-indexed family of action symbols.*

A semantic interpretation structure for a component signature is given by an algebra that interprets the sorts and operators, and a mapping for attributes and action symbols.

Definition 2.5.2. ***(Interpretation Structure)*** *A θ–interpretation structure for a signature $\theta = (\Sigma, A, \Gamma)$ is a triple $(\mathbb{U}, \mathbb{A}, \mathbb{G})$ where:*

- $\mathbb{U}$ *is a Σ-algebra, i.e., to each sort symbol $s \in S$ a set $s_{\mathbb{U}}$ is assigned, and to each function symbol $f \in \Omega_{\langle s_1, s_2, s_3, ..., s_n\rangle, s}$ a function $f_{\mathbb{U}} : s_{1\mathbb{U}} \times s_{2\mathbb{U}} \times ... \times s_{n\mathbb{U}} \to s_{\mathbb{U}}$ is assigned;*
- $\mathbb{A}$ *maps $f \in A_{\langle s_1, s_2, s_3, ..., s_n\rangle, s}$ to $\mathbb{A}(f) : s_{1\mathbb{U}} \times s_{2\mathbb{U}} \times ... \times s_{n\mathbb{U}} \times \mathbb{N} \to s_{\mathbb{U}}$*
- $\mathbb{G}$ *maps $g \in \Gamma_{\langle s_1, s_2, s_3, ..., s_n\rangle}$ to $\mathbb{G}(g) : s_{1\mathbb{U}} \times s_{2\mathbb{U}} \times ... \times s_{n\mathbb{U}} \to \wp(\mathbb{N})$.*

Definition 2.5.3. ***(loci)*** *Given an object signature $\theta = (\Sigma, A, \Gamma)$ and a θ–interpretation structure $(\mathbb{U}, \mathbb{A}, \mathbb{G})$, let*

$$E = \{i : \mathbb{N} \mid (\exists \vec{s} : S^*, g \in \Gamma_{\vec{s}}, \vec{a} : (\vec{s})_{\mathbb{U}} \bullet i \in \mathbb{G}(g)(\vec{a}))\}$$

That is, E consists of those instants during which an action of the object occurs, i.e. these instants denote state transitions in which the object will be engaged. Those state transitions are said to be witnessed by θ. The θ–interpretation structure $(\mathbb{U}, \mathbb{A}, \mathbb{G})$ is said to be a θ-locus *iff, for every instant $i \notin E$ and every $f \in A$, we have $\mathbb{A}(f)(i) = \mathbb{A}(f)(i + 1)$.*

The definition of terms and formulae are defined in the following sections. Before defining terms, we need to assume a collection ξ of distinct variables. For a given signature, a *classification* Ξ means a partial function $\xi \to S$ and $\Xi_s = \{x \in \xi \mid \Xi(x) = s\}$

Definition 2.5.4. ***(terms)*** *Given an object signature $\theta = (\Sigma, A, \Gamma)$ and a classification Ξ, S–indexed set of terms $TR_\theta(\Xi)$ are defined as follows:*

- *variables: if $\Xi(x) = s$, then x is a term in $TR_\theta(\Xi)_s$;*
- *$f \in \Omega_{\langle s_1,s_2,s_3,.....,s_n\rangle,s}$ and t_i are terms in $TR_\theta(\Xi)_{s_i}$ then $f(t_1, t_2, ..., t_n) \in TR_\theta(\Xi)_s$*
- *$f \in A_{\langle s_1,s_2,s_3,.....,s_n\rangle,s}$ and t_i are terms in $TR_\theta(\Xi)_{s_i}$ then $f(t_1, t_2, ..., t_n) \in TR_\theta(\Xi)_s$*
- *if $t \in TR_\theta(\Xi)_s$ then $(\mathbb{X}_t) \in TR_\theta(\Xi)_s$.*

Definition 2.5.5. ***(formulae)*** *Given a component signature θ and a classification Ξ, the atomic formulae over θ and Ξ are as follows:*

- *$g(t_1, ..., t_n)$ for $g \in \Gamma_{\langle s_1,s_2,s_3,.....,s_n\rangle}$, and t_i terms in $TR_\theta(\Xi)_{s_i}$*
- *$(t_1 =_s t_2)$ for t_1 and t_2 terms in $TR_\theta(\Xi)_s$ for some $s \in S$*
- ***Beg** (beginning of time)*

The usual propositional connectives ($\to$ stands for implication) and temporal operators $\mathbb{X}$,$\mathbb{F}$ are used as constructors of formulae.

Definition 2.5.6. ***(Locality Axiom)***

$$\left(\left(\bigvee_{g\in\Gamma}(\exists\, x_g \ \bullet\ g(x_g))\right) \vee \left(\bigwedge_{a\in A}(\forall\, x_a \ \bullet\ \mathbf{X}\, a(x_a) = a(x_a))\right)\right)$$

The "locality axiom" means that the value of the attributes of a component can only be changed by the actions declared for that component. Alternately, we can say that any of the actions will be executed, otherwise all the attribute values will remain unchanged.

The other way to represent locus is through locality axiom. Among the θ-interpretation structures for a signature $\theta = (\Sigma, A, \Gamma)$, those satisfying the locality axiom for θ (where x_g and x_a represent tuples of variables of the argument sorts of g, respectively a), are θ-locus.

Definition 2.5.7. ***(Component Specification)*** *The specification of a component is a pair (θ, Φ) where θ is the signature of the component and Φ is a finite set (can be empty) of formulae over θ and Ξ.*

2.5.2 System Specification

In this section, we shall explain how to combine several interconnected components (a collection of theories) to specify a more complex system specification. In order to serve this purpose, components specification are assembled as a node of a diagram, edges of the diagram represent the specification morphism (connection) between components and two independent components synchronize through a third component meaning, the third component describe the similarities between the components. This system specification can later be used to prove properties of the global system or use it as an integral part of a larger system.

A morphism between component specifications defines the notion of structure and establishes the relationship that must exist between different components so that one of them may be considered as a sub-component (sub-object) of the other. The signature morphism σ from θ_1 to θ_2 identifies the symbols in θ_2 that corresponds to the symbols in θ_1. The definition of a signature morphism are as follows:

Definition 2.5.8. ***(Signature Morphism)*** *Given two component signature $\theta_1 = (\Sigma_1, A_1, \Gamma_1)$ and $\theta_2 = (\Sigma_2, A_2, \Gamma_2)$, a* ***signature morphism*** *σ from θ_1 to θ_2 consists of*

- *A morphism of algebraic signature $\sigma_\nu = (\Sigma_1 \to \Sigma_2)$*
- *for each $f : s_1, ..., s_n \to s$ in A_1 an attribute $\sigma_\alpha(f) : \sigma_\nu(s_1), ..., \sigma_\nu(s_n) \to \sigma_\nu(s)$ in A_2,*
- *for each $g : s_1, ...s_n$ in Γ_1 an event symbol $\sigma_\gamma(g) : \sigma_\nu(s_1), ...\sigma_\nu(s_n)$ in Γ_2.*

For a given signature morphism $\sigma : \theta_1 \to \theta_2$, how the formulas are translated from one component to another is described as follows:

Definition 2.5.9. ***(Translation)*** *Given a signature morphism $\sigma : \theta_1 \to \theta_2$, the translation of terms formulas are defined as follows:*

- *If $x \in \Xi_s$ is a variable , then $\sigma(x) = x$*
- *If $f \in \Omega_{\langle s_1,.....,s_n\rangle,s}$ then $\sigma(f(t_1, ..., t_n)) = \sigma_\nu(f)(\sigma(t_1), ..., \sigma(t_n))$*
- *If $f \in A_{\langle s_1,.....,s_n\rangle,s}$ then $\sigma(f(t_1, ..., t_n)) = \sigma_\alpha(f)(\sigma(t_1), ..., \sigma(t_n))$*
- *If $g \in \Gamma_{\langle s_1,.....,s_n\rangle}$ then $\sigma(g(t_1, ..., t_n)) = \sigma_\gamma(g)(\sigma(t_1), ..., \sigma(t_n))$*
- $\sigma(\mathbb{X}\ t) = (\mathbb{X}\ \sigma(t))$
- $\sigma(t_1 = t_2) = (\sigma(t_1) = \sigma(t_2))$
- $\sigma(\mathbb{X}\ p) = (\mathbb{X}\ \sigma(p))$
- *and similar for other connectives.*

A morphism between two component specifications is a signature morphism σ such that for every valid formula $p \in \Phi_1$ the translation of the formula $\sigma(p)$ would be valid in Φ_2. A signature morphism does not necessarily preserve locality, that is the translation of the locality formula in Φ_1 does not necessarily hold in Φ_2. So, in order to be a morphism the mapping will have to preserve the locality as well. That is, for a given signature σ the translation of the locality axiom in (θ_1, Φ_1) would be a theorem in (θ_2, Φ_2). The translation of the locality axiom is abbreviated as specification morphism. The formal definition of specification morphism is as follows:

Definition 2.5.10. ***(Specification Morphism)*** *Given component descriptions (θ_1, Φ_1) and (θ_2, Φ_2), a specification morphism $\sigma : (\theta_1, \Phi_1) \to (\theta_2, \Phi_2)$ is a signature morphism $\sigma : \theta_1 \to \theta_2$ such that*

- *$(\Phi_2 \Rightarrow_{\theta_2} \sigma(p))$ is valid for every $p \in \Phi_1$,*
- *$(\Phi_2 \Rightarrow_{\theta_2} \theta_1 \xrightarrow{\sigma} \theta_2)$ is valid (locality preservation).*

Proposition 2.5.11. ***(Category DESC)*** *Component specifications and their specification morphisms (monomorphism) form a category DESC.*

Proposition 2.5.12. ***(Category MonoDESC)*** *Component specifications and their specification monomorphisms form a category MonoDESC.*

Definition 2.5.13. ***(System Architecture)*** *In light of the definition of a diagram (Definition 2.2.2), a system architecture is a diagram in the category MonoDESC (Proposition 2.5.11) with an acyclic shape graph.*

A directed cycle is a directed path whose first and last vertices are the same, and that includes at least one edge. A directed acyclic graph is a graph with no directed cycles. The shape graphs associated with system architectures are acyclic.

Definition 2.5.14. ***(Properties of a system architecture)*** *Properties of a system architecture are the properties of the colimit of the diagram, i.e., an agreement between stakeholders on how the system should behave.*

Definition 2.5.15. ***(SigHom, SpecHom)*** *Given two component signatures $\theta_1 = (\Sigma_1, A_1, \Gamma_1)$ and $\theta_2 = (\Sigma_2, A_2, \Gamma_2)$, and the* signature morphism *σ from θ_1 to θ_2, we write* SigHom *θ_1 θ_2 for the set of all* signature morphisms *from θ_1 to θ_2. Given two component specification (θ_1, ϕ_1) and (θ_2, ϕ_2), and the specification homomorphism σ from (θ_1, ϕ_1) to (θ_2, ϕ_2), we write* SpecHom *(θ_1, ϕ_1) (θ_2, ϕ_2) for the set of all specification morphisms from (θ_1, ϕ_1) to (θ_2, ϕ_2).*

Definition 2.5.16. ***(Identity SingHom)*** *The identity SigHom on θ is denoted by $SigId_\theta$ and the* identity SpecHom *on (θ, ϕ) is denoted by $SpecId_{(\theta,\phi)}$.*

Component specifications and their morphisms constitute a category. Intuitively, in order to get the system specification we will have to take the colimit (amalgamated sum) of the system architecture (a diagram in *MonoDESC*). Here, edges represent the sub-object/sub-component relationship between nodes. Two nodes synchronize by sharing a sub-component, and the sub-component identifies the commonalities between two components and assists communication between them. An edge between two nodes a and b means that a is a sub-object of b and b is a super-object of a.

3 Zigzag Matching for Join Point Patterns

This chapter elaborates on the research question and introduces the general technique to approach this problem. In section 3.2, by considering a concrete example, we demonstrate that the kind of aspect introduction we aim for is not covered by existing graph transformation concepts. The following sections formally define all the terminology associated with the formal description of our methodology.

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*.

3.1 Introduction of Running Example (Use Case)

Consider the diagram in Figure 3.1.1. This diagram is an architecture for sender-receiver communication. Here the component *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* $\xrightarrow{TR2R}$. For a complete case study review appendix B.

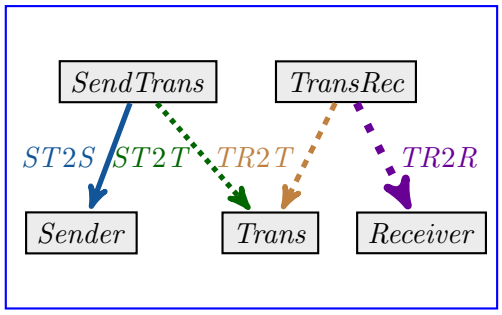


Figure 3.1.1: Unsecured communication

Now, consider the architecture in Figure 3.1.2. 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.

So, what did we do here? We have 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

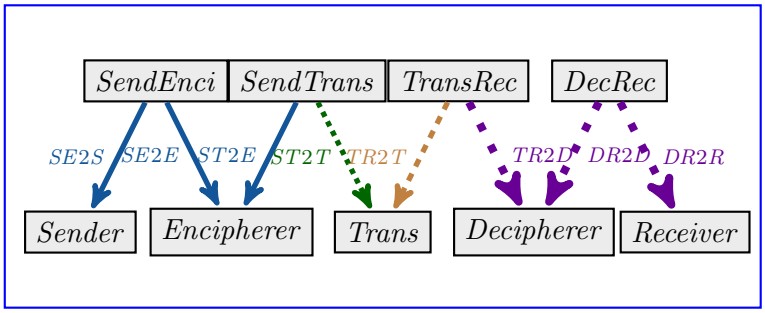


Figure 3.1.2: Secured communication

software architectures? Simple cases of this, as the one shown in Figure 3.1.3, suggest that standard categorical graph transformation concepts, such as the double-pushout (DPO) approach explained by Corradini et al. (1997), should be applicable. Different style and color edges are used in the diagrams to make the matching obvious.

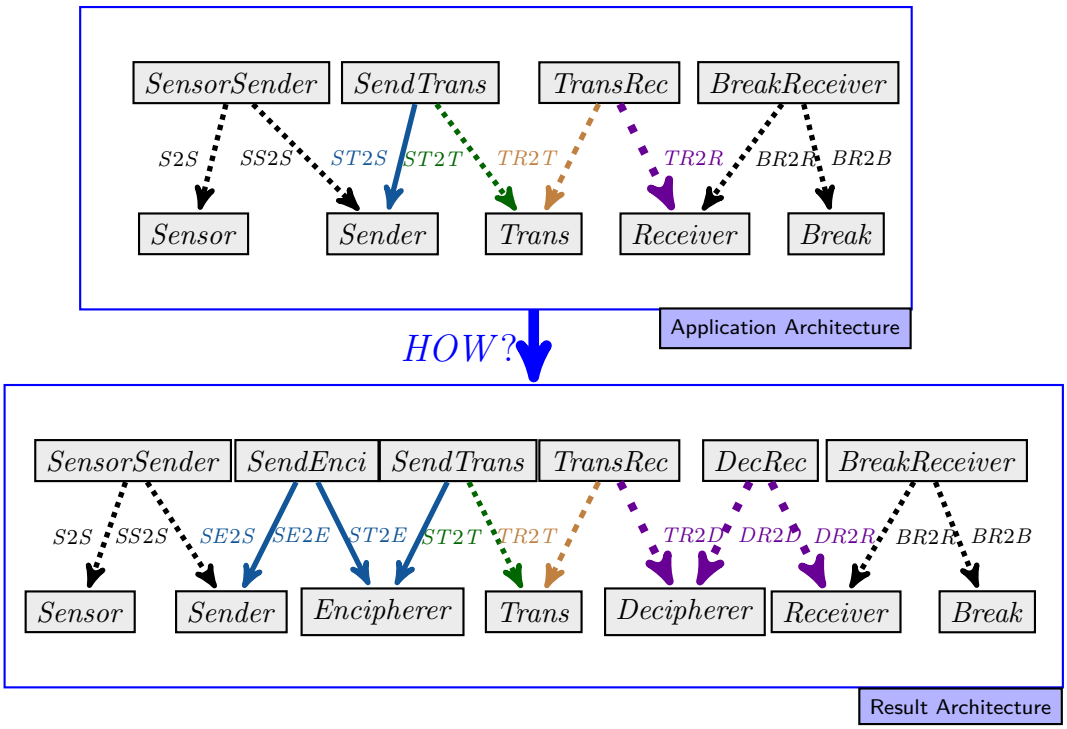


Figure 3.1.3: Introducing security in communication

As depicted in Figure 3.1.3, for a bigger/more complex architecture where sender-receiver communication exists, how can we introduce a security aspect to this architecture and make the architecture "secured"? The simple and powerful technique we shall apply as an approach to our problem is the graph transformation.

3.2 Exploration of Established Transformation Techniques

None of the established (i.e., set-theoretic, algebraic) graph transformation technique is applicable in our case. As an example, both the double-pushout and the single-pushout approaches work on a single edge but the type of graph transformation we require works on some sort of path, i.e., zigzag path. In the following sections, with an example, we analyze the application of the DPO (Double-pushout) and the SPO (Single-pushout) approaches to introduce two aspects, namely security and reliability, into a sender-receiver communication architecture.

3.2.1 Introduce Aspects by DPO Approach

Suppose there is an architecture where the sender-receiver communication exists and we want to introduce security/reliability/security-reliability to the given architecture by applying the double-pushout approach. As a very first step, we have to define the two production rules, corresponding to the two aspects.

3.2.1.1 Rule: DPO Security Introduction

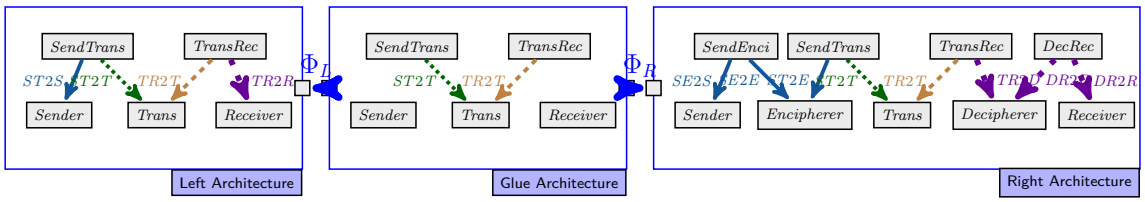


Figure 3.2.1: Rule: DPO Security Introduction

An annotation of the production rule "DPO Security Introduction" in the Figure 3.2.1 is as follows:

- Left Architecture side is the precondition of the rule
- Right Architecture side is the post condition of the rule
- Edges to be deleted are $ST2S$ and $TR2R$ ("Left Architecture" ← "Glue Architecture")
- "Right Architecture" → "Glue Architecture" describes the part to be created.

3.2.1.2 Rule: DPO Reliability Introduction:

An annotation of the production rule "DPO Reliability Introduction" in the Figure 3.2.2 is as follows:

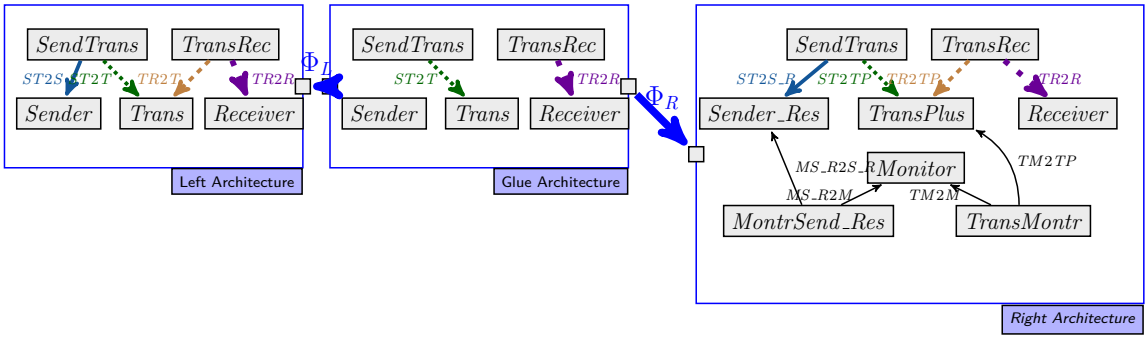


Figure 3.2.2: Rule: DPO Reliability Introduction

For a given bigger architecture, i.e, an "Application Architecture", if there exists a matching of the "Left Architecture" into the "Application Architecture", in "Result Architecture" the image of Trans would have the additional material of TransPlus addes to it, and the associated image of the edges $ST2S$ and $TR2T$ would be adapted according to the edge $ST2S\_R$ and $TR2TP$ respectively. All other edges ($ST2T, TR2R$ images), having pre-images in the "Glue Architecture" via the "Left Architecture", will remain unchanged. All newly added nodes in the "Right Architecture" will also remain unchanged, edges incident with Sender_Res and *TransPlus* will be adapted to fit the images in the "Application Architecture".

3.2.1.3 Reliability Aspect Introduction: DPO Approach

For given an architecture where sender-receiver communication exists, if we want to introduce reliability into it through DPO approach, the transformation will look like as follows:

The step by step annotation of the above transformation is as follows:

- $\mathcal{X}_L$ is the structure preserving matching from "Left-hand side" to "Application Architecture"
- Secondly, we construct the "Host Architecture" ("Application Architecture" − "Left-hand side" ∪ Ξ ("Glue Architecture"))
- After that, we check for "gluing" condition
- Finally, we construct the pushout for "Glue Architecture", "Host Architecture", "Right-hand side" and the two morphisms Ξ and Φ_R.

3.2.1.4 Example: Why DPO is Not Applicable?

Now, for given an architecture where secured but unreliable communication exists, if we want to introduce reliability into this architecture through DPO approach and

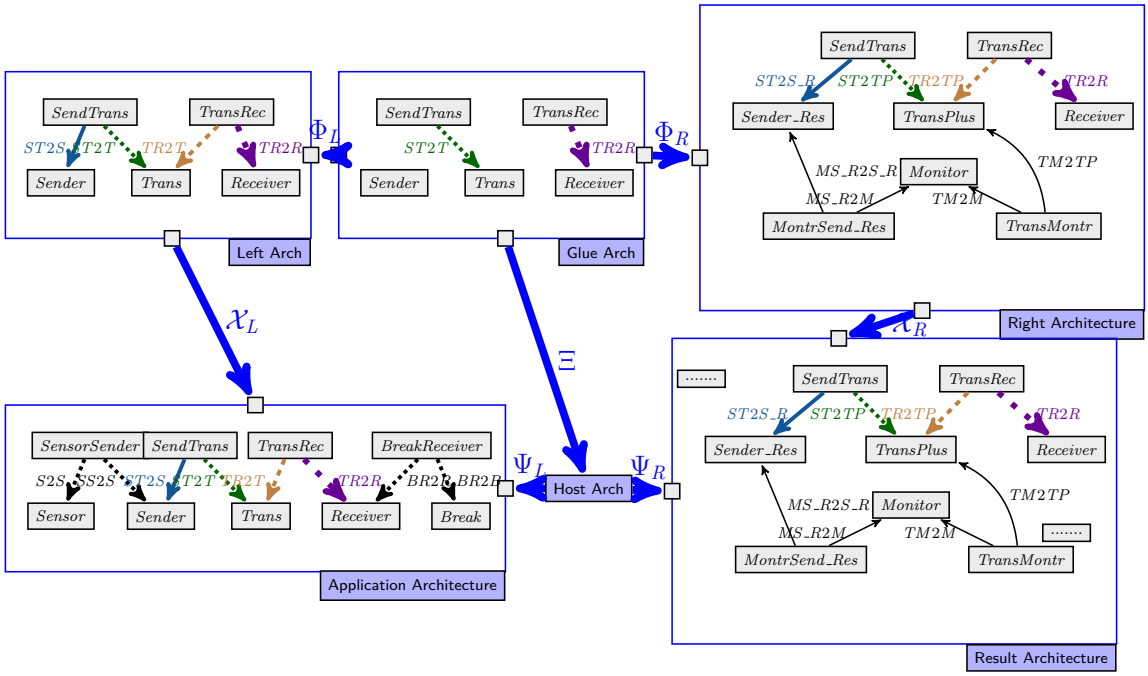


Figure 3.2.3: Introduce Reliability by DPO Approach

make the architecture both secure and reliable, the transformation will look like as follows:

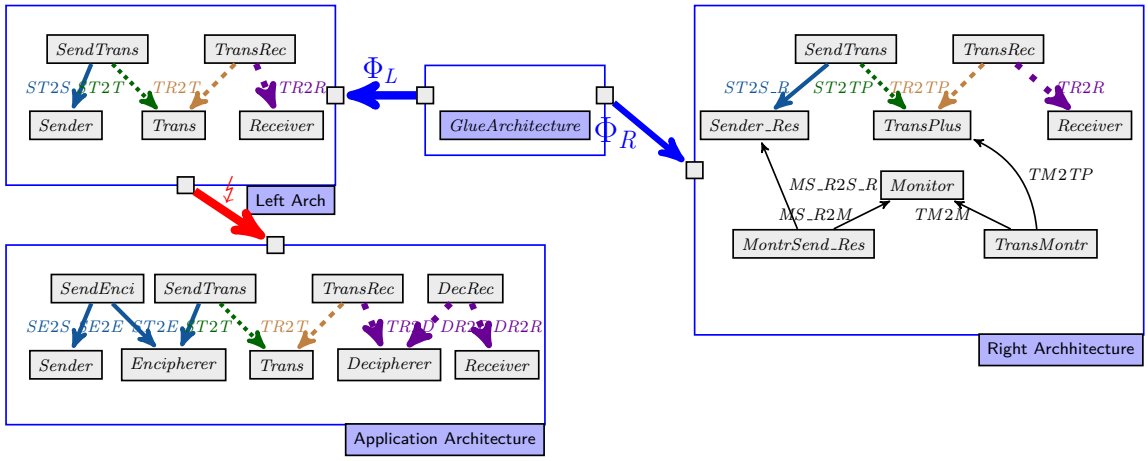


Figure 3.2.4: Endeavor to introduce reliability in secured communication by DPO

However, we notice that there is no structure-preserving matching $\mathcal{X}_L$ between "Left-hand side" to "Application Architecture". In short, $\mathcal{X}_L$ maps Sender to Sender and Receiver to Receiver, but we can not find any match for $ST2S$ and $TR2T$ in our "Application Architecture". Hence, in this kind of setting the conventional graph

homomorphism does not work and therefore the DPO approach does not work. For the same reason, the "single-pushout" approach does not work either.

3.3 Zigzag Path

A directed path is a sequence of vertices connected by directed edges where all the edges are traversed along their direction. The graphs we are using are directed, but the paths are not, i.e., edges in a path could be traversed in any directions.

Definition 3.3.1. ***(Zigzag Path)*** *A zigzag path in a graph $\mathscr{G} = (\mathscr{V}, \mathscr{E}, src, tgt)$ is an alternating nonempty list $\langle v_0, e_1, v_1, e_2, v_2, ...e_k, v_k\rangle$ of vertices and directed edges traversed in arbitrary direction*
where,

- *$k > 0$ is the length of p.*
- *e_i is incident with v_{i-1} and v_i for $i \in 1..., k$,*

We define: (with $p = \langle v_0, e_1, v_1, e_2, v_2, ...e_k, v_k\rangle$)

- *$\mathscr{Z}path_{\mathscr{G}}$ is the collection of $\mathscr{Z}path$ paths in $\mathscr{G}$*
- *$source : \mathscr{Z}path_{\mathscr{G}} \to \mathscr{V}$ is the source function for $\mathscr{Z}path_{\mathscr{G}}$ with $source(p) = v_0$*
- *$target : \mathscr{Z}path_{\mathscr{G}} \to \mathscr{V}$ is the target function for $\mathscr{Z}path_{\mathscr{G}}$ with $target(p) = v_k$*
- *$fstEdge : \mathscr{Z}path_{\mathscr{G}} \to \mathscr{E}$ with $fstEdge(p) = e_1$*
- *$lstEdge : \mathscr{Z}path_{\mathscr{G}} \to \mathscr{E}$ with $lstEdge(p) = e_k$*
- *$length : \mathscr{Z}path_{\mathscr{G}} \to \mathbb{N}$*

For given *zigzag* paths $p = \langle v_0, e_1, v_1, e_2, v_2, ...e_k, v_k\rangle$ and $q = \langle v_k, e_{k+1}, v_{k+1}, e_{k+2}, v_{k+2}, ..., v_m\rangle$, the concatenation is defined as: $p \ddagger q = \langle v_0, e_1, v_1, e_2, v_2, ..., v_m\rangle$

3.4 Zigzag Graph Transformation: The Matching

In order to face the challenge, we are introducing a new kind of transformation technique. The type of transformation we need is somewhat related to hyperedge replacement graph grammars, but has a different organization and restricted subgraphs. Here, a single edge can be mapped to a zigzag path explained in Section 3.3 and mappings between vertices are specification homomorphism. Our morphisms could be considered as hyperedge replacement steps where the hyperedges are binary, and the replacement graphs are restricted to a simple undirected path.

The mapping between the underlying shape graphs of two system architectures is called zigzag graph homomorphism, which we have treated in Section 3.5.1. The

mapping between two system architectures is defined as system architecture zigzag homomorphism. The system architecture zigzag homomorphism defines a mapping between two system architectures, where in the underlying shape graphs single edge maps to a zigzag path by mapping the source node of the edge to the source node of the zigzag path and the target node of the edge to the target node of the zigzag path. In the system architecture level, there exists an identity specification homomorphism from the specification of the edge's source nodes to the specification of the zigzag path's source node. But, the morphism between the specification of the edge's target node to the specification of the zigzag path's target node is a specification homomorphism. For more details and precise definition see Section 3.7.3.

The type of matching we require is pictured in Section 3.4.1; the fact that we need this kind of "indirect" matching is the most obvious reason why conventional DPO/SPO is not directly applicable.

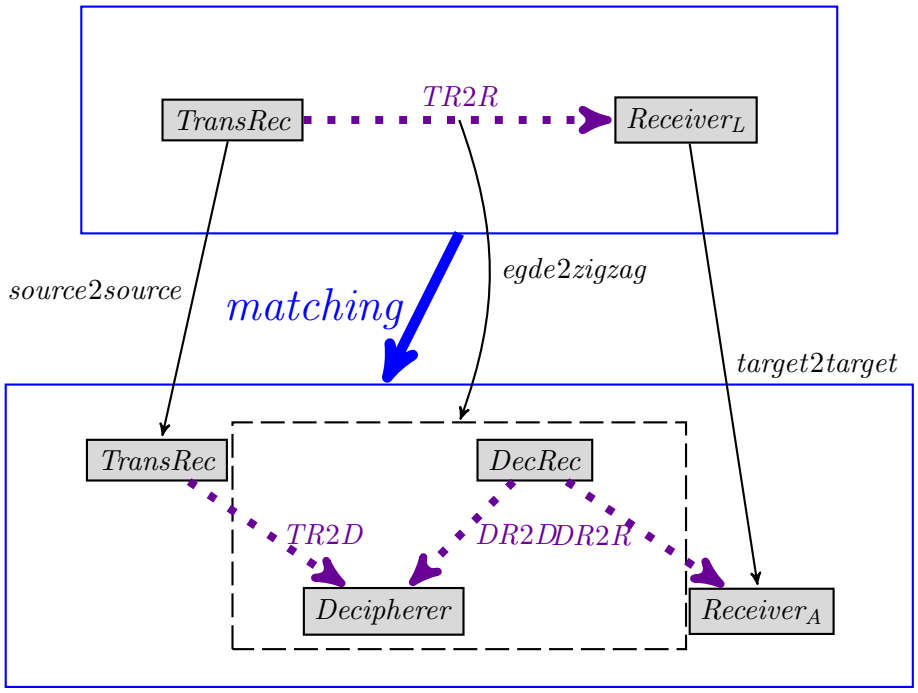
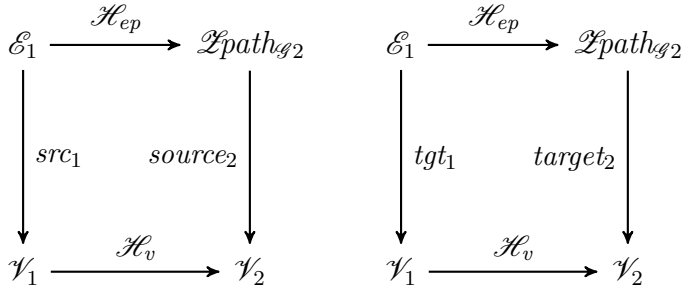


Figure 3.4.1: Matching single edge to undirected path

In Figure 3.4.1, the edge $TransRec \rightarrow Receive_L$ maps to a zigzag path $TransRec \rightarrow Decipherer \leftarrow DecRec \rightarrow Receiver_A$. Here, the morphism $source2souce$ maps $TransRec$ to $TransRec$ and is an identity specification homomorphism. On the other hand, the morphism $target2target$ maps $Receiver_L$ to $Reciver_A$ and is a *specification homomorphism* from $Receiver_L$ to $Reciver_A$.

3.5 The category ZigzagGraphs of graphs

Definition 3.5.1. ***(Zigzag Graph Homomorphism)*** *Given two graphs $\mathscr{G}_i = (\mathscr{V}_i, \mathscr{E}_i, src_i, tgt_i)$ for $i \in \{1, 2\}$, a zigzag graph homomorphism $\mathscr{H} : \mathscr{G}_1 \to \mathscr{G}_2$ consists of two functions, $\mathscr{H}_v : \mathscr{V}_1 \to \mathscr{V}_2$, $\mathscr{H}_{ep} : \mathscr{E}_1 \to \mathscr{Z}path_{\mathscr{G}_2}$ such that the following two diagrams commute:*



We will need a special kind of mapping where each *edge* is mapped to the *zigzag path* whose length is 1, i.e., for $p \in \mathscr{Z}path_{\mathscr{G}}$, length$(p) = 1$, and the direction of the edge is preserved in the *zigzag path*.

Definition 3.5.2. ***(Unit Zigzag Mapping)*** *The* Unit zigzag mapping $\iota : \mathscr{E} \to \mathscr{Z}path_{\mathscr{G}}$ *is the function that maps each edge e to the* zigzag path *consisting only of e, namely: $\langle src(e), e, tgt(e)\rangle$*

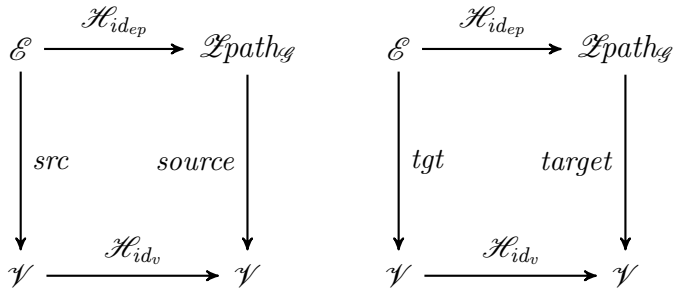
We denote the category ZigzagGraphs of graphs as follows. Here,

- The *objects* are graphs
- The *morphisms* are zigzag graph homomorphisms
- For every object $\mathscr{G} \in$ Obj(ZigzagGraphs), the *identity morphism* is $id_{\mathscr{G}} : \mathscr{G} \to \mathscr{G}$, the *identity zigzag graph homomorphism*.

Definition 3.5.3. ***(Identity Zigzag Graph Homomorphism)*** *Given a graph $\mathscr{G} = (\mathscr{V}, \mathscr{E}, src, tgt)$, the identity zigzag graph homomorphism $\mathscr{H}_{id_{\mathscr{G}}} : \mathscr{G} \to \mathscr{G}$ are defined as $\mathscr{H}_{id_v}(v) = id$ and $\mathscr{H}_{id_{ep}}(e) = (src(e), e, tgr(e))$ such that the following diagrams commute:*

Now we need to define the *composition* of *zigzag graph homomorphisms*. In order to define that, let us define a function $\mathscr{Z}map$ first which can be understood as Kleisli composition with respect to the path monad.

Definition 3.5.4. ***($\mathscr{Z}map$)*** $\mathscr{Z}map : (\mathscr{H}_{12} : \mathscr{G}_1 \to \mathscr{G}_2) \to \mathscr{Z}path_{\mathscr{G}1} \to \mathscr{Z}path_{\mathscr{G}2}$
$\mathscr{Z}map\ \mathscr{H}_{12}\langle v_0, e_1, v_1\rangle = \mathscr{H}_{12ep}(e_1)$
$\mathscr{Z}map\ \mathscr{H}_{12}\langle up \ddagger_1 uq\rangle = \mathscr{Z}map\ \mathscr{H}_{12}(up) \ddagger_2 \mathscr{Z}map\ \mathscr{H}_{12}(uq)$



For three graphs $\mathscr{G}_i = (\mathscr{V}_i, \mathscr{E}_i, src_i, tgt_i)$ for $i \in \{1, 2, 3\}$ and two *zigzag graph homomorphism* $\mathscr{H}_{i(i+1)} : \mathscr{G}_i \to \mathscr{G}_{i+1}$ for $i \in \{1, 2\}$, the composition of the two *zigzag graph homomorphisms* is defined as $\mathscr{H}_{13} = \mathscr{H}_{23} \circ \mathscr{H}_{12}$ means $\mathscr{H}_{13v} = \mathscr{H}_{23v} \circ \mathscr{H}_{12v}$ and $\mathscr{H}_{13ep} = \mathscr{Z}map\ \mathscr{H}_{23} \circ \mathscr{H}_{12ep}$

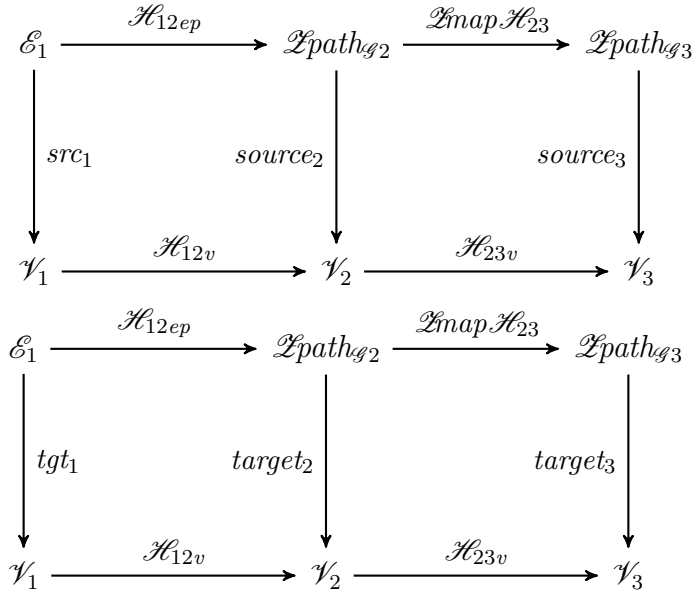


Figure 3.5.1: Well-definedness of $\mathscr{H}_{13}$

Proof of well-definedness:

The next step is for well-definedness of $\mathscr{H}_{13}$, to prove that all the inner squares commute. According to the definition of *zigzag graph homomorphism* we can claim that the left squares commute. Now we need to prove by induction that, the right squares commute.

Base Step:

$$
\begin{aligned}
& source_3 \circ \mathscr{Zmap}\ \mathscr{H}_{23}\langle v_0, e_1, v_1\rangle \\
= \quad & \langle \text{Definition of } \mathscr{Zmap}\rangle \\
& source_3 \circ \mathscr{H}_{23ep}\ (e_1) \\
= \quad & \langle \text{Definition zigzag hom}\rangle \\
& \mathscr{H}_{23v} \circ src_2\ (e_1) \\
= \quad & \langle \text{zigzag path length 1}\rangle \\
& \mathscr{H}_{23v} \circ source_2\langle v_0, e_1, v_1\rangle
\end{aligned}
$$

Induction Step: The L.H.S. is:

$$
\begin{aligned}
& source_3 \circ \mathscr{Zmap}\ \mathscr{H}_{23}\langle v_0, e_1, v_1, ...e_m, v_m\rangle \\
= \quad & \langle \text{IH}\rangle \\
& \mathscr{H}_{23v} \circ source_2\langle v_0, e_1, v_1, ...e_m, v_m\rangle
\end{aligned}
$$

Now we need to prove that:

$$
\begin{aligned}
& source_3 \circ \mathscr{Zmap}\ \mathscr{H}_{23}\langle v_0, e_1, v_1, ...e_m, v_m, e_{m+1}, v_{m+1}\rangle \\
= \quad & \mathscr{H}_{23v} \circ source_2\langle v_0, e_1, v_1, ...e_m, v_m, e_{m+1}, v_{m+1}\rangle
\end{aligned}
$$

Proof:

$$
\begin{aligned}
& source_3 \circ \mathscr{Zmap}\ \mathscr{H}_{23}\langle v_0, e_1, v_1, ...e_m, v_m, e_{m+1}, v_{m+1}\rangle \\
= \quad & \langle \text{Concatenation}\rangle \\
& (source_3 \circ \mathscr{Zmap}\ \mathscr{H}_{23})(\langle v_0, e_1, v_1, ...e_m, v_m\rangle \ddagger_2 \langle v_m, e_{m+1}, v_{m+1}\rangle) \\
= \quad & \langle \mathscr{Zmap}\ \ \text{Definition}\rangle \\
& source_3(\mathscr{Zmap}\ \mathscr{H}_{23}\langle v_0, e_1, v_1, ...e_m, v_m\rangle \ddagger_3 \mathscr{Zmap}\ \mathscr{H}_{23}\langle v_m, e_{m+1}, v_{m+1}\rangle) \\
= \quad & \langle \text{Simplification}\rangle \\
& source_3(\mathscr{Zmap}\ \mathscr{H}_{23}\langle v_0, e_1, v_1, ...e_m, v_m)\rangle \\
= \quad & \langle \text{Function composition}\rangle \\
& (source_3 \circ \mathscr{Zmap}\ \mathscr{H}_{23})\langle v_0, e_1, v_1, ...e_m, v_m\rangle \\
= \quad & \langle \text{IH}\rangle \\
& (\mathscr{H}_{23v} \circ source_2)\langle v_0, e_1, v_1, ...e_m, v_m\rangle \\
= \quad & \langle \text{Function composition}\rangle \\
& \mathscr{H}_{23v} \circ source_2\langle v_0, e_1, v_1, ...e_m, v_m, e_{m+1}, v_{m+1}\rangle
\end{aligned}
$$

Composition of zigzag graph homomorphism:

Since both inner squares of the top diagram in the Figure 3.5.1 commute, the outer square also commute. That is:

$$
\begin{aligned}
& source_3 \circ \mathscr{Z}map\ \mathscr{H}_{23} \circ \mathscr{H}_{12ep} \\
= \quad & \langle \text{Associativity} \rangle \\
& (source_3 \circ \mathscr{Z}map\ \mathscr{H}_{23}) \circ \mathscr{H}_{12ep} \\
= \quad & \langle source_3 \circ \mathscr{Z}map\ \mathscr{H}_{23} = \mathscr{H}_{23v} \circ source_2 \rangle \\
& \mathscr{H}_{23v} \circ source_2 \circ \mathscr{H}_{12ep} \\
= \quad & \langle source_2 \circ \mathscr{H}_{12ep} = \mathscr{H}_{12v} \circ src_1 \rangle \\
& \mathscr{H}_{23v} \circ \mathscr{H}_{12v} \circ src_1
\end{aligned}
$$

Similarly we can prove that the other outer square in the Figure 3.5.1 also commutes. Since all the squares commutes, it is proven that the composition of two *zigzag graph homomorphisms* is a *zigzag graph homomorphism*, i.e., $\mathscr{H}_{23} \circ \mathscr{H}_{12} = \mathscr{H}_{13}$

Identity law:

For any arbitrary *zigzag graph homomorphism* $\mathscr{H}_{12} : \mathscr{G}_1 \to \mathscr{G}_2$ the composition of $\mathscr{H}_{12} \circ \mathscr{H}_{id_{\mathscr{G}_1}}$ is $\mathscr{H}_{12}$:

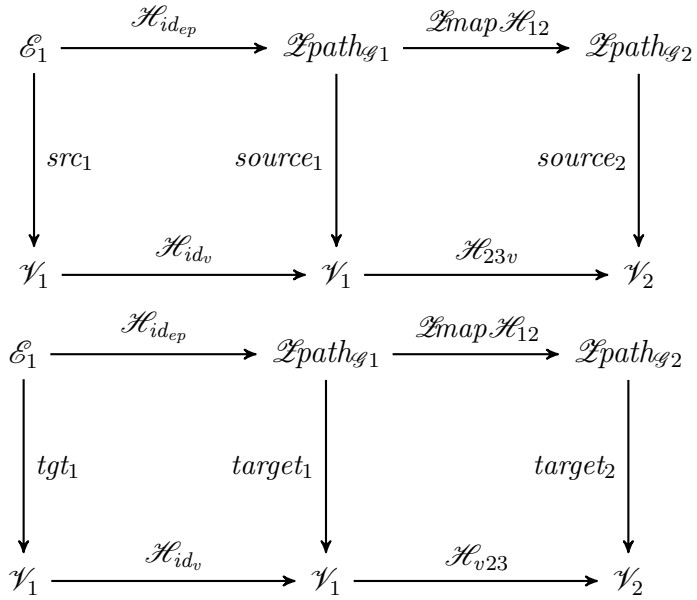


Figure 3.5.2: Identity Law

Since the outer squares commutes, $\mathscr{H}_{12} \circ \mathscr{H}_{id_{\mathscr{G}_1}} = \mathscr{H}_{12}$. Similarly we can prove that $\mathscr{H}_{id_{\mathscr{G}_2}} \circ \mathscr{H}_{12} = \mathscr{H}_{12}$. The associativity law also clearly holds. So, *ZigzagGraphs*

is a category.

3.6 Relational Homomorphism

The relation-algebraic notation introduced in this section is useful to define some category. The relation-algebraic notation along with those category are useful for some proofs in section 3.7 and section 4.3. For further details, please follow Schmidt and Ströhlein (1993), Kahl (2002), and Kahl (2004).

Definition 3.6.1 (Relational zigzag graph homomorphism)**.** *For two given graphs* $\mathscr{G}_i = (\mathscr{V}_i, \mathscr{E}_i, src_i, tgt_i)$ *for* $i \in \{1, 2\}$*, a* relational zigzag graph homomorphism $\mathscr{H}_{ZZ} : \mathscr{G}_1 \leftrightarrow \mathscr{G}_2$ *consists of two relations,* $\mathscr{H}_v : \mathscr{V}_1 \leftrightarrow \mathscr{V}_2$*,* $\mathscr{H}_{ep} : \mathscr{E}_1 \leftrightarrow \mathscr{Z}path_{\mathscr{G}_2}$ *such that they satisfy the inclusion* ${\mathscr{H}_{ep}}^{\smile} ; src_1 \sqsubseteq source_2 ; {\mathscr{H}_v}^{\smile}$ *and* ${\mathscr{H}_{ep}}^{\smile} ; tgt_1 \sqsubseteq target_2 ; {\mathscr{H}_v}^{\smile}$ *represented by the sub-commuting diagrams in the Figure 3.6.1.*

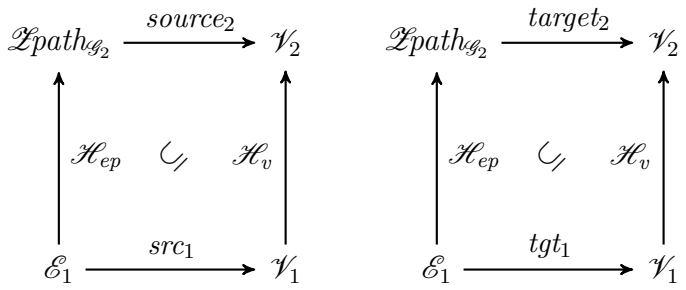


Figure 3.6.1: Relational zigzag graph homomorphism

Proposition 3.6.2. *A univalent and total* relational zigzag graph homomorphism *is a* zigzag graph homomorphism.

Definition 3.6.3. ***(Category** RelZigzagGraphs**)** Graphs and their relational zigzag graph homomorphism form a category which will be called RelZigzagGraphs.*

Definition 3.6.4. ***(Functor** RelGraphsToRelZigzgGraphs**)** RelGraphsToRelZpathGraphs is a functor between RelGraphs and RelZigzagGraphs category, denoted by* $\mathsf{F}_{\mathsf{GRZG}}$*.*

When in Section 3.7 and 4.4.2 we write something like
$\mathscr{U}_R = \Psi^{\smile} ; \Psi' \cup \chi^{\smile} ; \chi'$
it means that, $\mathscr{U}_R$ is a relational zigzag graph homomorphism, and since χ' and Ψ' are zigzag graph homomorphism, hence they are relational zigzag graph homomorphism as well. χ and Ψ are graph homomorphism, so we can consider them as a relational graph homomorphism. Since there is a functor between *RelGraphs* and *RelZigzagGraphs*, $\chi^{\smile}$ and $\Psi^{\smile}$ are relational zigzag graph homomorphism as well.

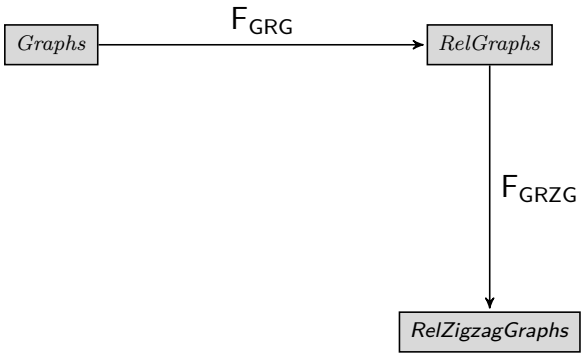


Figure 3.6.2: Relationship between *Graphs*, *RelGraphs*, *RelZigzagGraphs*

3.7 Pushout

Theorem 3.7.1. *In* ZigzagGraphs*, a pushout of a graphs span in Graphs is a* ZigzagGraphs *pushout too.*

Proof: In *ZigzagGraphs*, the object $\mathscr{P}$ along with the morphisms $\chi : \mathscr{R} \to \mathscr{P}$ and $\Psi : \mathscr{A} \to \mathscr{P}$, is a pushout in *Graphs* for the morphisms $\Phi : \mathscr{L} \to \mathscr{R}$ and $\Xi : \mathscr{L} \to \mathscr{A}$.

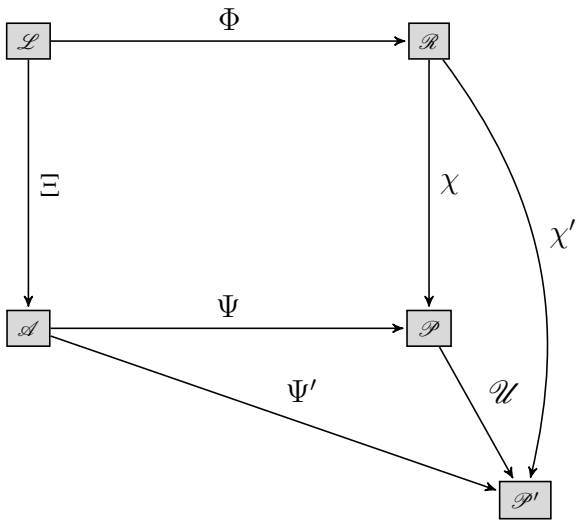


Figure 3.7.1: ZigzagGraphs pushout

For any other given cocone $\mathscr{A} \xrightarrow{\Psi'} \mathscr{P}' \xleftarrow{\chi'} \mathscr{R}$ in *ZigzagGraphs* where $\Phi \fatsemi \chi' = \Xi \fatsemi \Psi'$, we first define a relational zigzag graph morphism $\mathscr{U}_R : \mathscr{P} \leftrightarrow \mathscr{P}'$ by $\mathscr{U}_R = \Psi^{\smile} \fatsemi \Psi' \cup \chi^{\smile} \fatsemi \chi'$. This satisfies $\chi \fatsemi \mathscr{U}_R = \chi'$ and, $\Psi \fatsemi \mathscr{U}_R = \Psi'$.

Uniqueness:

Assume, there is another zigzag graph morphism $\mathscr{V}_R : \mathscr{P} \to \mathscr{P}'$ such that $\chi \,;\mathscr{V}_R = \chi'$ and, $\Psi \,;\mathscr{V}_R = \Psi'$. Now we need to show that $\mathscr{V}_R = \mathscr{U}_R$. From the definition we have:

$$
\begin{array}{ll}
 & \mathscr{U}_R \\
= & \quad \langle \text{Definition} \rangle \\
 & \Psi^{\smile} \,;\Psi' \cup \chi^{\smile} \,;\chi' \\
= & \quad \langle \text{Substitute } \chi' \text{ and } \Psi' \rangle \\
 & \Psi^{\smile} \,;\Psi \,;\mathscr{V}_R \cup \chi^{\smile} \,;\chi \,;\mathscr{V}_R \\
= & \quad \langle \text{Distributive Law} \rangle \\
 & (\Psi^{\smile} \,;\Psi \cup \chi^{\smile} \,;\chi) \,;\mathscr{V}_R \\
= & \quad \langle \Psi \text{ and } \chi \text{ are univalent and jointly surjective, PO } \rangle \\
 & \mathbb{I} \,;\mathscr{V}_R \\
= & \quad \langle \text{Identity Law} \rangle \\
 & \mathscr{V}_R
\end{array}
$$

So, it is shown that $\mathscr{U}_R$ is unique.

$\mathscr{U}_R$ is a *zigzag graph morphism*:

In order to show that relational zigzag graph morphism $\mathscr{U}_R$ is a zigzag graph morphism $\mathscr{U}$, we need to show that $\mathscr{U}_R$ is total and univalent.

Total:

$\mathscr{U}_R$ is total iff $\mathscr{U}_{\mathscr{V}}$ total and $\mathscr{U}_{\mathscr{E}}$ both are total.

$$
\begin{array}{ll}
 & \mathscr{U}_{\mathscr{V}} \text{ total} \\
= & \quad \langle \text{Definition of total} \rangle \\
 & \mathbb{I} \subseteq \mathscr{U}_{\mathscr{V}} \,;{\mathscr{U}_{\mathscr{V}}}^{\smile}
\end{array}
$$

$$
\begin{array}{ll}
\textit{Proof}: & \quad \langle \text{Right Hand Side} \rangle \\
 & \mathscr{U}_{\mathscr{V}} \,;{\mathscr{U}_{\mathscr{V}}}^{\smile} \\
= & \quad \langle \text{Definition} \rangle \\
 & (\Psi_{\mathscr{V}}{}^{\smile} \,;\Psi'_{\mathscr{V}} \cup \chi_{\mathscr{V}}{}^{\smile} \,;\chi'_{\mathscr{V}}) \,;(\Psi'_{\mathscr{V}}{}^{\smile} \,;\Psi_{\mathscr{V}} \cup \chi'_{\mathscr{V}}{}^{\smile} \,;\chi_{\mathscr{V}}) \\
= & \quad \langle \text{Distribute } ; \text{ over } \cup \rangle \\
 & \Psi_{\mathscr{V}}{}^{\smile} \,;\Psi'_{\mathscr{V}} \,;\Psi'_{\mathscr{V}}{}^{\smile} \,;\Psi_{\mathscr{V}} \cup \chi_{\mathscr{V}}{}^{\smile} \,;\chi'_{\mathscr{V}} \,;\Psi'_{\mathscr{V}}{}^{\smile} \,;\Psi_{\mathscr{V}} \cup \Psi_{\mathscr{V}}{}^{\smile}; \\
 & \Psi'_{\mathscr{V}} \,;\chi'_{\mathscr{V}}{}^{\smile} \,;\chi_{\mathscr{V}} \cup \chi_{\mathscr{V}}{}^{\smile} \,;\chi'_{\mathscr{V}} \,;\chi'_{\mathscr{V}}{}^{\smile} \,;\chi_{\mathscr{V}}
\end{array}
$$

$$
\begin{array}{ll}
\supseteq & \langle \Psi'_{\mathscr{V}} \text{ and } \chi'_{\mathscr{V}} \text{ are total} \rangle \\
& \Psi_{\mathscr{V}}{}^{\smile} \,;\, \Psi_{\mathscr{V}} \cup \chi_{\mathscr{V}}{}^{\smile} \,;\, \chi_{\mathscr{V}} \\
\supseteq & \langle \text{PO} \rangle \\
& \mathbb{I}
\end{array}
$$

Now we show that $\mathscr{U}_{\mathscr{E}}$ is total. That is

$$
\begin{array}{ll}
& \mathscr{U}_{\mathscr{E}} \text{ total} \\
= & \langle \text{Definition of total} \rangle \\
& \mathbb{I} \subseteq \mathscr{U}_{\mathscr{E}} \,;\, \mathscr{U}_{\mathscr{E}}{}^{\smile}
\end{array}
$$

$$
\begin{array}{lll}
\textit{Proof} : & & \langle \text{Right Hand Side} \rangle \\
& & \mathscr{U}_{\mathscr{E}} \,;\, \mathscr{U}_{\mathscr{E}}{}^{\smile} \\
& = & \langle \text{Definition} \rangle \\
& & (\Psi_{\mathscr{E}}{}^{\smile} \,;\, \Psi'_{\mathscr{E}} \cup \chi_{\mathscr{E}}{}^{\smile} \,;\, \chi'_{\mathscr{E}}) \,;\, (\Psi'_{\mathscr{E}}{}^{\smile} \,;\, \Psi_{\mathscr{E}} \cup \chi'_{\mathscr{E}}{}^{\smile} \,;\, \chi_{\mathscr{E}}) \\
& = & \langle \text{Distribute } ; \text{ over } \cup \rangle \\
& & \Psi_{\mathscr{E}}{}^{\smile} \,;\, \Psi'_{\mathscr{E}} \,;\, \Psi'_{\mathscr{E}}{}^{\smile} \,;\, \Psi_{\mathscr{E}} \cup \chi_{\mathscr{E}}{}^{\smile} \,;\, \chi'_{\mathscr{E}} \,;\, \Psi'_{\mathscr{E}}{}^{\smile} \,;\, \Psi_{\mathscr{E}} \cup \Psi_{\mathscr{E}}{}^{\smile} \,;\, \Psi'_{\mathscr{E}} \,;\, \chi'_{\mathscr{E}}{}^{\smile} \,;\, \chi_{\mathscr{E}} \cup \\
& & \chi_{\mathscr{E}}{}^{\smile} \,;\, \chi'_{\mathscr{E}} \,;\, \chi'_{\mathscr{E}}{}^{\smile} \,;\, \chi_{\mathscr{E}} \\
& \supseteq & \langle \Psi'_{\mathscr{E}} \text{ and } \chi'_{\mathscr{E}} \text{ are total} \rangle \\
& & \Psi_{\mathscr{E}}{}^{\smile} \,;\, \Psi_{\mathscr{E}} \cup \chi_{\mathscr{E}}{}^{\smile} \,;\, \chi_{\mathscr{E}} \\
& \supseteq & \langle \text{PO} \rangle \\
& & \mathbb{I}
\end{array}
$$

Ψ and χ are jointly surjective so $\Psi^{\smile}$ and $\chi^{\smile}$ would be jointly total. Since, Ψ' and χ' are already total, $\mathscr{U}$ is also total.

Univalent:

$\mathscr{U}_R$ is univalent iff $\mathscr{U}_{\mathscr{V}}$ and $\mathscr{U}_{\mathscr{E}}$ both are univalent.

$$
\begin{array}{ll}
& \mathscr{U}_{\mathscr{V}} \text{ univalent} \\
= & \langle \text{Definition of univalent} \rangle \\
& \mathscr{U}_{\mathscr{V}}{}^{\smile} \,;\, \mathscr{U}_{\mathscr{V}} \subseteq \mathbb{I}
\end{array}
$$

$$
\begin{array}{lll}
\textit{Proof} : & & \langle \text{Left Hand Side} \rangle \\
& & \mathscr{U}_{\mathscr{V}}{}^{\smile} \,;\, \mathscr{U}_{\mathscr{V}} \\
& = & \langle \text{Definition} \rangle \\
& & (\Psi'_{\mathscr{V}}{}^{\smile} \,;\, \Psi_{\mathscr{V}} \cup \chi'_{\mathscr{V}}{}^{\smile} \,;\, \chi_{\mathscr{V}}) \,;\, (\Psi_{\mathscr{V}}{}^{\smile} \,;\, \Psi'_{\mathscr{V}} \cup \chi_{\mathscr{V}}{}^{\smile} \,;\, \chi'_{\mathscr{V}})
\end{array}
$$

$$
\begin{array}{ll}
= & \langle \text{Distribute } ; \text{ over } \cup \rangle \\
& \Psi'_{\mathscr{Y}}{}^{\smile};\ \Psi_{\mathscr{Y}};\ \Psi_{\mathscr{Y}}{}^{\smile};\ \Psi'_{\mathscr{Y}} \cup \Psi'_{\mathscr{Y}}{}^{\smile};\ \Psi_{\mathscr{Y}};\ \chi_{\mathscr{Y}}{}^{\smile};\ \chi'_{\mathscr{Y}} \cup \\
& \chi'_{\mathscr{Y}}{}^{\smile};\ \chi_{\mathscr{Y}};\ \Psi_{\mathscr{Y}}{}^{\smile};\ \Psi'_{\mathscr{Y}} \cup \chi'_{\mathscr{Y}}{}^{\smile};\ \chi_{\mathscr{Y}};\ \chi_{\mathscr{Y}}{}^{\smile};\ \chi'_{\mathscr{Y}} \\
\subseteq & \left\langle\ \Psi_{\mathscr{Y}} \text{ and } \chi_{\mathscr{Y}} \text{ are injective, } \Psi_{\mathscr{Y}};\Psi_{\mathscr{Y}}{}^{\smile} \subseteq \mathbb{I},\ \chi_{\mathscr{Y}};\chi_{\mathscr{Y}}{}^{\smile} \subseteq \mathbb{I}\ \right\rangle \\
& \Psi'_{\mathscr{Y}}{}^{\smile} ;\ \Psi'_{\mathscr{Y}} \cup \Psi'_{\mathscr{Y}}{}^{\smile} ;\ \Psi_{\mathscr{Y}} ;\ \chi_{\mathscr{Y}}{}^{\smile} ;\ \chi'_{\mathscr{Y}} \cup \chi'_{\mathscr{Y}}{}^{\smile} ;\ \chi_{\mathscr{Y}} ;\ \Psi_{\mathscr{Y}}{}^{\smile} ;\ \Psi'_{\mathscr{Y}} \cup \chi'_{\mathscr{Y}}{}^{\smile} ;\ \chi'_{\mathscr{Y}} \\
\subseteq & \left\langle\ \mathscr{W} : \mathscr{A} \leftrightarrow \mathscr{R},\ \text{PO},\ \mathscr{W}_{\mathscr{Y}} = \Psi_{\mathscr{Y}};\ \chi_{\mathscr{Y}}{}^{\smile}\ \right\rangle \\
& \Psi'_{\mathscr{Y}}{}^{\smile};\ \Psi'_{\mathscr{Y}} \cup \Psi'_{\mathscr{Y}}{}^{\smile};\ \mathscr{W}_{\mathscr{Y}};\chi'_{\mathscr{Y}} \cup \chi'_{\mathscr{Y}}{}^{\smile};\ \mathscr{W}^{\smile}_{\mathscr{Y}};\Psi'_{\mathscr{Y}} \cup \chi'_{\mathscr{Y}}{}^{\smile};\ \chi'_{\mathscr{Y}} \\
\subseteq & \left\langle\ \mathscr{W}_{\mathscr{Y}};\chi'_{\mathscr{Y}} \subseteq \Psi'_{\mathscr{Y}},\ \mathscr{W}^{\smile}_{\mathscr{Y}};\ \Psi'_{\mathscr{Y}} \subseteq \chi'_{\mathscr{Y}}\ \right\rangle \\
& \Psi'_{\mathscr{Y}}{}^{\smile};\ \Psi'_{\mathscr{Y}} \cup \Psi'_{\mathscr{Y}}{}^{\smile};\ \Psi'_{\mathscr{Y}} \cup \chi'_{\mathscr{Y}}{}^{\smile};\ \chi'_{\mathscr{Y}} \cup \chi'_{\mathscr{Y}}{}^{\smile};\ \chi'_{\mathscr{Y}} \\
\subseteq & \langle \text{Idempotent Law} \rangle \\
& \Psi'_{\mathscr{Y}}{}^{\smile} ;\ \Psi'_{\mathscr{Y}} \cup \chi'_{\mathscr{Y}}{}^{\smile} ;\ \chi'_{\mathscr{Y}} \\
\subseteq & \langle\ \Psi'_{\mathscr{Y}} \text{ and } \chi'_{\mathscr{Y}} \text{ are univalent } \rangle \\
& \mathbb{I}
\end{array}
$$

Now we show that $\mathscr{U}_{\mathscr{E}}$ is univalent. That is

$$
\begin{array}{ll}
& \mathscr{U}_{\mathscr{E}} \text{ univalent} \\
= & \langle \text{Definition of univalent} \rangle \\
& \mathscr{U}_{\mathscr{E}}{}^{\smile} ;\ \mathscr{U}_{\mathscr{E}} \subseteq \mathbb{I}
\end{array}
$$

We show this inclusion starting from the left-hand side:

$$
\begin{array}{ll}
& \mathscr{U}_{\mathscr{E}}{}^{\smile} ;\ \mathscr{U}_{\mathscr{E}} \\
= & \langle \text{Definition} \rangle \\
& (\Psi'_{\mathscr{E}}{}^{\smile} ;\ \Psi_{\mathscr{E}} \cup \chi'_{\mathscr{E}}{}^{\smile} ;\ \chi_{\mathscr{E}}) ;\ (\Psi_{\mathscr{E}}{}^{\smile} ;\ \Psi'_{\mathscr{E}} \cup \chi_{\mathscr{E}}{}^{\smile} ;\ \chi'_{\mathscr{E}}) \\
= & \langle \text{Distribute } ; \text{ over } \cup \rangle \\
& \Psi'_{\mathscr{E}}{}^{\smile} ;\ \Psi_{\mathscr{E}} ;\ \Psi_{\mathscr{E}}{}^{\smile} ;\ \Psi'_{\mathscr{E}} \cup \Psi'_{\mathscr{E}}{}^{\smile} ;\ \Psi_{\mathscr{E}} ;\ \chi_{\mathscr{E}}{}^{\smile} ;\ \chi'_{\mathscr{E}} \cup \\
& \chi'_{\mathscr{E}}{}^{\smile} ;\ \chi_{\mathscr{E}} ;\ \Psi_{\mathscr{E}}{}^{\smile} ;\ \Psi'_{\mathscr{E}} \cup \chi'_{\mathscr{E}}{}^{\smile} ;\ \chi_{\mathscr{E}} ;\ \chi_{\mathscr{E}}{}^{\smile} ;\ \chi'_{\mathscr{E}} \\
\subseteq & \left\langle\ \Psi_{\mathscr{E}},\ \chi_{\mathscr{E}} \text{ are injective, } \Psi_{\mathscr{E}};\Psi_{\mathscr{E}}{}^{\smile} \subseteq \mathbb{I},\ \chi_{\mathscr{E}};\chi_{\mathscr{E}}{}^{\smile} \subseteq \mathbb{I}\right\rangle \\
& \Psi'_{\mathscr{E}}{}^{\smile};\ \Psi'_{\mathscr{E}} \cup \Psi'_{\mathscr{E}}{}^{\smile};\ \Psi_{\mathscr{E}};\ \chi_{\mathscr{E}}{}^{\smile};\ \chi'_{\mathscr{E}} \cup \chi'_{\mathscr{E}}{}^{\smile};\ \chi_{\mathscr{E}};\ \Psi_{\mathscr{E}}{}^{\smile};\ \Psi'_{\mathscr{E}} \cup \chi'_{\mathscr{E}}{}^{\smile};\ \chi'_{\mathscr{E}} \\
\subseteq & \left\langle\ \mathscr{W}_{\mathscr{E}} : \mathscr{A} \leftrightarrow \mathscr{R},\ \text{PO},\ \mathscr{W}_{\mathscr{E}} = \Psi_{\mathscr{E}};\ \chi_{\mathscr{E}}{}^{\smile}\right\rangle \\
& \Psi'_{\mathscr{E}}{}^{\smile};\ \Psi'_{\mathscr{E}} \cup \Psi'_{\mathscr{E}}{}^{\smile};\ \mathscr{W}_{\mathscr{E}};\chi'_{\mathscr{E}} \cup \chi'_{\mathscr{E}}{}^{\smile};\ \mathscr{W}^{\smile}_{\mathscr{E}};\Psi'_{\mathscr{E}} \cup \chi'_{\mathscr{E}}{}^{\smile};\ \chi'_{\mathscr{E}} \\
\subseteq & \left\langle\ \mathscr{W}_{\mathscr{E}};\ \chi'_{\mathscr{E}} \subseteq \Psi'_{\mathscr{E}},\ \mathscr{W}^{\smile}_{\mathscr{E}};\ \Psi'_{\mathscr{E}} \subseteq \chi'_{\mathscr{E}}\right\rangle \\
& \Psi'_{\mathscr{E}}{}^{\smile};\ \Psi'_{\mathscr{E}} \cup \Psi'_{\mathscr{E}}{}^{\smile};\ \Psi'_{\mathscr{E}} \cup \chi'_{\mathscr{E}}{}^{\smile};\ \chi'_{\mathscr{E}} \cup \chi'_{\mathscr{E}}{}^{\smile};\ \chi'_{\mathscr{E}} \\
\subseteq & \langle \text{Idempotent Law} \rangle \\
& \Psi'_{\mathscr{E}}{}^{\smile} ;\ \Psi'_{\mathscr{E}} \cup \chi'_{\mathscr{E}}{}^{\smile} ;\ \chi'_{\mathscr{E}} \\
\subseteq & \langle\ \Psi'_{\mathscr{E}} \text{ and } \chi'_{\mathscr{E}} \text{ are univalent } \rangle \\
& \mathbb{I}
\end{array}
$$

Ψ and χ are injective so $\Psi^{\smile}$ and $\chi^{\smile}$ are univalent. Given that, Ψ' and χ' are already univalent and there is a relation $\mathscr{W} : \mathscr{A} \leftrightarrow \mathscr{R}$, $\mathscr{U}$ is univalent.

Hence, $\mathscr{U}_R$ is a *zigzag graph homomorphism* $\mathscr{U}$.

So, in *ZigzagGraphs* if we can find a pushout of a graphs span in *Graphs*, it would be a pushout for *ZigzagGraphs* too.

3.7.1 Path Relation $PRel_\sigma$

Definition 3.7.2 (Path Relation)**.** *For two given component signatures* $\theta_1 = (\Sigma_1, A_1, \Gamma_1)$ *and* $\theta_2 = (\Sigma_2, A_2, \Gamma_2)$*, and a* signature homomorphism $\sigma : \theta_1 \to \theta_2$ *with* $\sigma = (\sigma_\nu, \sigma_\alpha, \sigma_\gamma)$*; the* signature relation Rel_σ *between* $(\theta_1 \leftrightarrow \theta_2)$ *is the tuple* $(\sigma_\nu, \sigma_\alpha, \sigma_\gamma)$ *consisting of the three functions defined in* σ *considered as relation.*

For a given zigzag path p *in diagram* D *the* path relation $PRel(D, p)$ *is a relational specification homomorphism between* $D(source(p))$ *and* $D(target(p))$ *defined as follows:*

1. *(Empty Path:) For a given empty path* v_0*,*

$$PRel\langle v_0\rangle = Rel_\sigma(\mathbb{I}_{v_0})$$

2. *(Single Edge* $\rightarrow$*:) For a path* $\langle v_0, e_1, v_1\rangle$*, with a single forward edge* $e_1 : v_0 \to v_1$

$$PRel\langle v_0, e_1, v_1\rangle = Rel_\sigma(e_1)$$

3. *(Single Edge* $\leftarrow$*:) For a path* $\langle v_0, e_1, v_1\rangle$*, with a single backward edge* $e_1 : v_0 \to v_1$

$$PRel\langle v_0, e_1, v_1\rangle = Rel_\sigma(e_1)^{\smile}$$

4. *(Arbitrary Path:) For given an arbitrary path* $\langle v_0, e_1, v_1, p\rangle$*,*

$$PRel\langle v_0, e_1, v_1, p\rangle = PRel\langle v_0, e_1, v_1\rangle \fatsemi PRel\langle v_1, p\rangle$$

Definition 3.7.3 (**Category SysArchs**)**.** *The* SysArchs *is a category where objects are system architectures and morphisms are system architecture homomorphisms.*

3.7.2 System Architecture Homomorphism: The Simple Case/(edge map)

A simple *system architecture homomorphism* $\mathscr{H} : \mathscr{A}_1 \to \mathscr{A}_2$ form 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:

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

This means that the diagram 3.7.2 commutes:

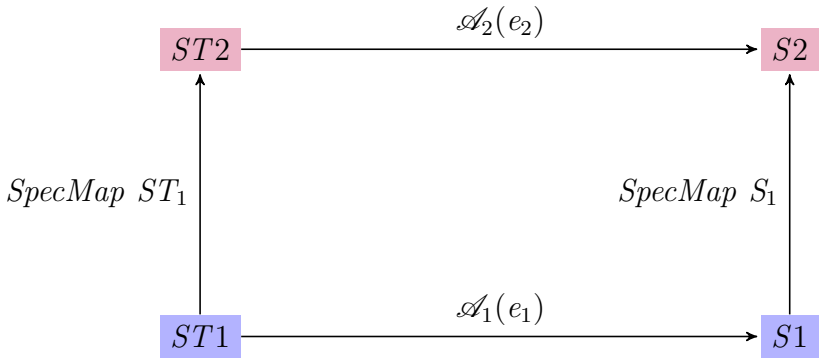


Figure 3.7.2: System Architecture Homomorphism

3.7.3 System Architecture Zigzag Homomorphism: The General Case/(zigzag map)

In the general case where a singe edge in $\mathscr{A}_1.\mathscr{E}$ may be mapped to a zigzag path $\mathscr{A}_2.\mathscr{Z}path$, a system architecture zigzag homomorphism is a tuple $\mathscr{H} = (\mathscr{H}_{\mathscr{V}}, \mathscr{H}_{\mathscr{Z}}, \mathscr{H}_{SpecMap}, PRel_{\sigma}$ 3.
consisting of:

- node mapping $\mathscr{H}_{\mathscr{V}} : \mathscr{A}_1.\mathscr{V} \to \mathscr{A}_2.\mathscr{V}$,
- zigzag mapping $\mathscr{H}_{\mathscr{Z}} : \mathscr{A}_1.\mathscr{E} \to \mathscr{A}_2.\mathscr{Z}path$,
- 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:

$$\begin{array}{rcl} \mathscr{A}_2.source(\mathscr{H}_{\mathscr{Z}}\ e) & = & \mathscr{H}_{\mathscr{V}}(\mathscr{A}_1.src\ e) \\ \mathscr{A}_2.target(\mathscr{H}_{\mathscr{Z}}\ e) & = & \mathscr{H}_{\mathscr{V}}(\mathscr{A}_1.tgt\ e) \\ \mathscr{H}_{SpecMap}(src\ e) & \text{is an isomorphism} & \end{array}$$

And we can assert some relationship between *source* and *target* nodes of a zigzag path at specification level. This relationship would be useful for property preservation. Having no relation between *source* and *target* specification requires regorous proof, but having some relation helps to prove properties by inspection. For further details see chapter 5.

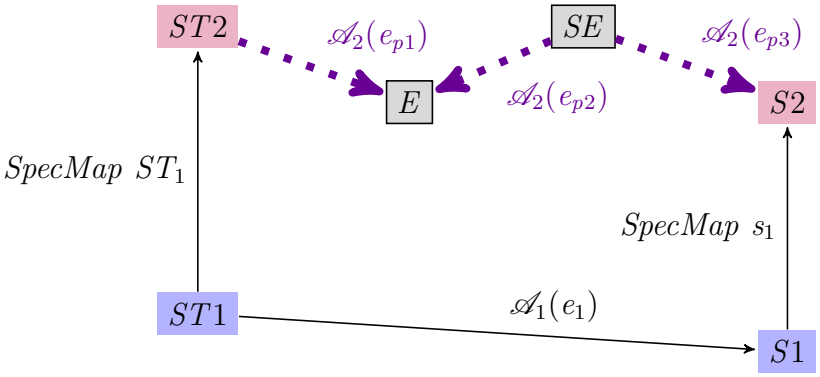


Figure 3.7.3: System Architecture Zigzag Homomorphism (Bottom,Up)

In the case of an incomplete (some items are related) path relation, we may define predicates, e.g., sort preserving, operator preserving, attribute preserving and/or action preserving, to put a constraint and define the relations to reflect one's particular need. In our case, the path relation could be *Empty* (no relation), *Complete* (all items are related) or in-between without constraining any predicates.

3.7.4 The Category SysArchsZ

Given a system architecture $\mathscr{A}$, the identity system architecture zigzag homomorphism $\mathscr{H}_{id_{\mathscr{A}}} : \mathscr{A} \to \mathscr{A}$ is the identity system architecture homomorphism, $\mathscr{H}_{id_{\mathscr{A}}} = (\mathscr{H}_{\mathscr{V}}, \mathscr{H}_{\mathscr{E}}, \mathscr{H}_{SpecMap})$ where
$\mathscr{H}_{\mathscr{V}}(n) = \mathbb{I}$,
$\mathscr{H}_{\mathscr{E}}(e) = \mathscr{H}_{\mathscr{Z}}(e) = (src(e), e, tgt(e))$,
$\mathscr{H}_{SpecMap}\ n = SpecId\ \mathscr{A}\ n$

We define the category *SysArchsZ* as follows. Here,

- The *objects* are System Architectures.
- The *morphisms* are system architecture zigzag homomorphisms.

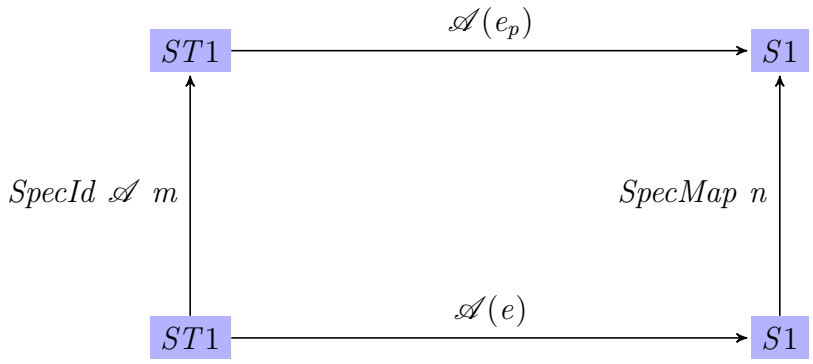


Figure 3.7.4: Identity System Architecture Homomorphism

- For every object $\mathscr{A} \in Ob(SysArchsZ)$, the identity morphism $id_{\mathscr{A}} : \mathscr{A} \to \mathscr{A}$, is the identity system architecture zigzag homomorphism.

The morphism between two shape or index graphs is defined as zigzag graph homomorphism. According to the definition of the *ZigzagGraphs* category, the composition of two zigzag graph homomorphisms is a zigzag graph homomorphism and the composition satisfies the identity and associativity law. Moreover, the system architecture zigzag homomorphism does not introduce any new commutativity. Therefore, the composition of two system architecture zigzag homomorphism would be a system architecture zigzag homomorphism and would satisfy the identity and associativity law. Hence, *SysArchsZ* is a category.

4 Construction of the Resulting Architecture through Zigzag Diagram Transformation

For the system architectures as defined in the previous chapter, 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. In the following sections, we will then find an appropriate formalization for the kind of diagram transformation we need.

4.1 Zigzag Graph Transformation

For a defined set of production rules (e.g., security introduction, reliability introduction defined in Section 3.2.1) and given an "Application Architecture," if we apply our transformation technique, in all cases except one, the "Result Architecture" we obtain after the transformation is a pushout object. If aspect introduction relaxed an edge in "Right Architecture," and the matching of the edge in "Application Architecture" is also a relaxed zigzag path, then the category *SysArchsZ* does not have a pushout. In this case, the transformation is semi-automatic. The designer will have to make a design decision and depending on the decision the designer makes the "Result Architecture" could even be a pushout object.

4.1.1 Effect of Aspect Introduction: All Cases

Generally, an aspect introduction may modify a single edge in different ways. A couple of the possible ways are pictured in the Figure 4.1.1. Here a single edge $S \to T$ is modified by four different ways (Case One-Four), but neither of them is practical. One of the main reason for them not being practical is that the underlying graph of a system architecture is a *bipartite graph*. But, in this example, except for the original edge, neither of the modified graphs is a bipartite graph, i.e., they violated the *connector-component alternation* pattern. In the original edge, S is a part of a connector and T is a component. But, in One) both S and RTS are connector parts in Two) either S and SP are part of a connector or SP and T are components. Similarly, the case Three and Four are also violating the property of a bipartite graph. Another reason for a modified edge being an impractical one is: either they can be represented by one of the practical cases, or whatever the new zigzag path is expressing are already captured by the old edge. As an example, case Two could be represented as $S \to T$ and case Three could be represented as $S \to SP$. All the practical ways that an aspect introduction can modify a single edge are demonstrated in the following Section 4.1.2.

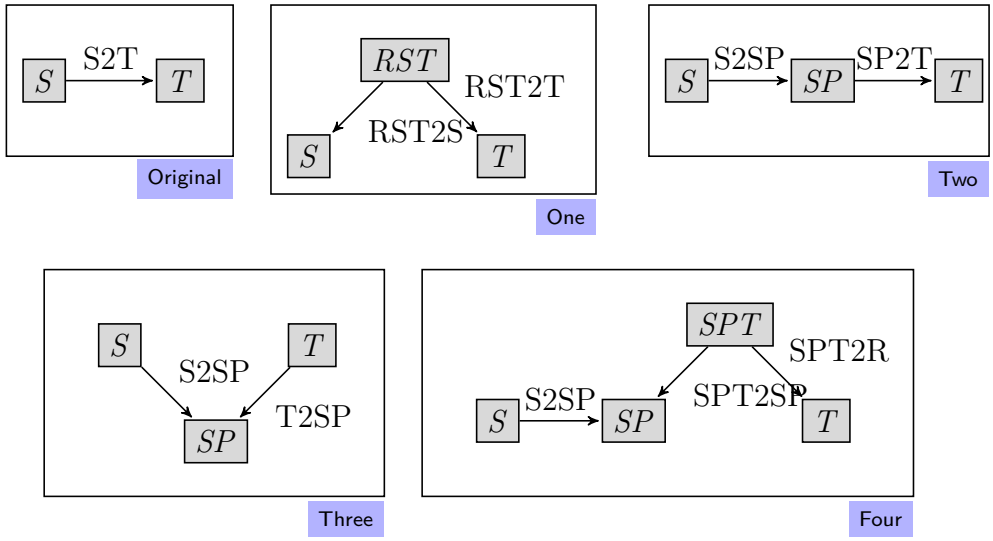


Figure 4.1.1: General Modification of a Single Edge

4.1.2 Effect of Aspect Introduction: Practical Cases

An aspect introduction can modify a single edge by only four practical ways, and they are pictured in the following Figure 4.1.2:

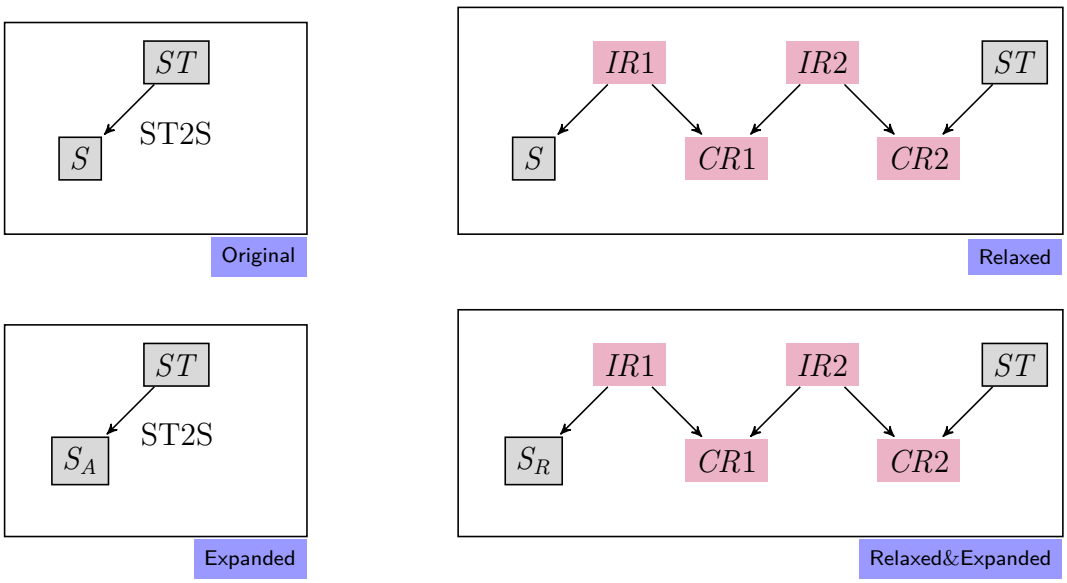


Figure 4.1.2: Practical Modification of a Single Edge

- The edge could remain unchanged.
- It could be relaxed by adding a couple of components and connectors between the source and target of the edge.

- The edge could be extended by expanding the component.
- It could be both relaxed and extended.

4.2 Construction of The Result Architecture: All Possibilities

The previous Section explained all four practical ways how aspect introduction modifies a single edge to a zigzag path. There are four considerable ways, i.e., **unchanged-wild/ exact-wild** (simple case), **expanded-expanded** (component expansion), **relaxing-relaxing** (component addition) and **relaxing-expanded/expanded-relaxing**, to distribute the above four possibilities between application and right-hand side architecture, and exploring those explains all the distribution possibilities. In all cases except the **relaxing-relaxing** one, the "result architecture" is a pushout construct in *SysArchsZ* category. In **relaxing-relaxing** (component addition both ways, see section 4.2.4) case the "result architecture" is not a pushout construct, but the construction is a systematic one. Depending on the distributions, how the transformation behaves is described in the following sections.

4.2.1 Component expansion both ways:

In this case, an edge is extended in both "Application" and "Right Architecture." The "Result Architecture" is a pushout construct in the category *SysArchsZ*.

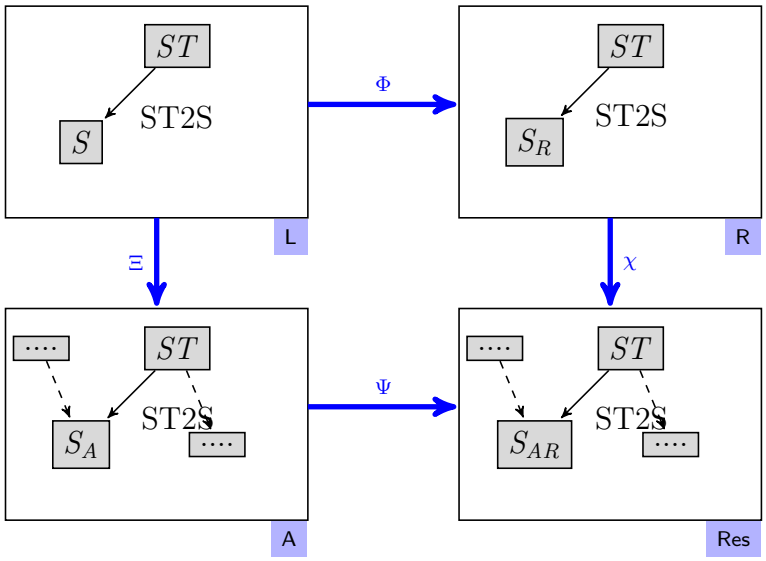


Figure 4.2.1: Component expansion both ways

In more details, the component S_{AR} in the "Result Architecture" is a pushout construct of the cone $S_A \xleftarrow{\Xi} S \xrightarrow{\Phi} S_R$ but the connector ST in the "Result Architecture"

is a direct copy, i.e., ST is identical in all architectures.

4.2.2 Component addition / identity matching:

If we can find a proper matching (exact matching) of an edge in the "Application Architecture"; in the "Result Architecture," we will replace this edge with the matched zigzag edge (Relaxed, Extended, Relaxed-Extended) from "Right Architecture." This "Result Architecture" is a pushout construct in the category *SysArchsZ*.

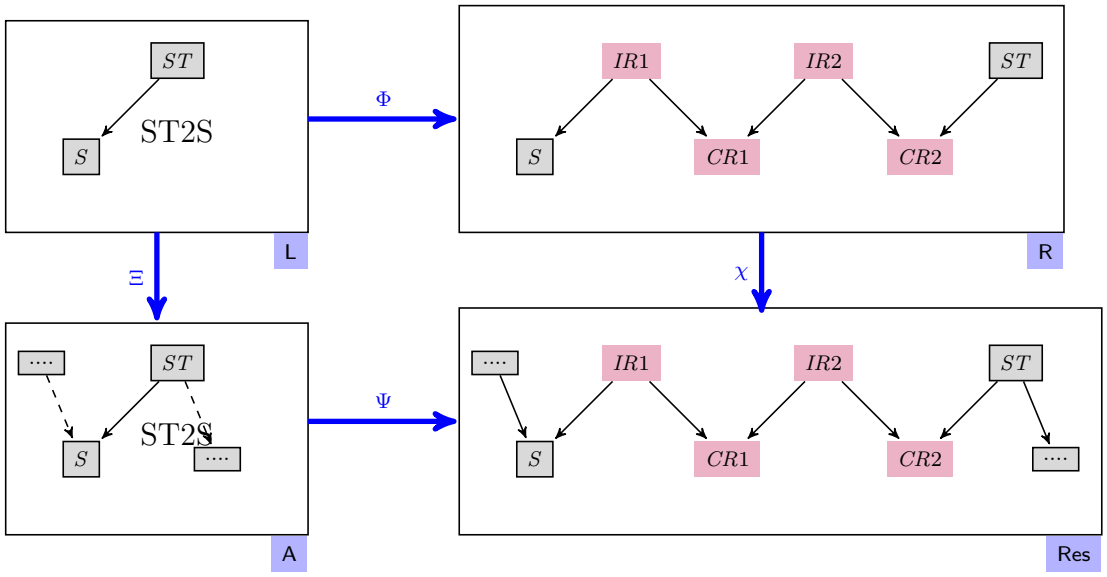


Figure 4.2.2: Component addition / identity matching

Figure 4.2.2 illustrates one of the simple cases where we have an exact match in the "Application Architecture" and the edge is relaxed in the "Right Architecture." So, in "Result Architecture" we have replaced the edge $ST \rightarrow S$ with the relaxed zigzag edge. The following cases illustrate how the transformation technique behaves if we do not find an exact matching in both "Application" and "Right Architecture."

4.2.3 Component addition / component expansion:

In this case, an edge is extended in one architecture, e.g., "Application Architecture," by extending (adding new property) the component, but is relaxed in other architecture, e.g., "Right Architecture," by adding a couple of new components and connectors as depicted in Figure 4.2.3. Even in this case, the "result architecture" is a pushout construct in the category *SysArchsZ*.

To construct the components and connectors in the "Result Architecture," we have to keep in mind some important points. In details, components having a preimage in the "Left Architecture" through the "Application" and the "Result Architecture" are constructed as a pushout construct. Components having preimages either in the

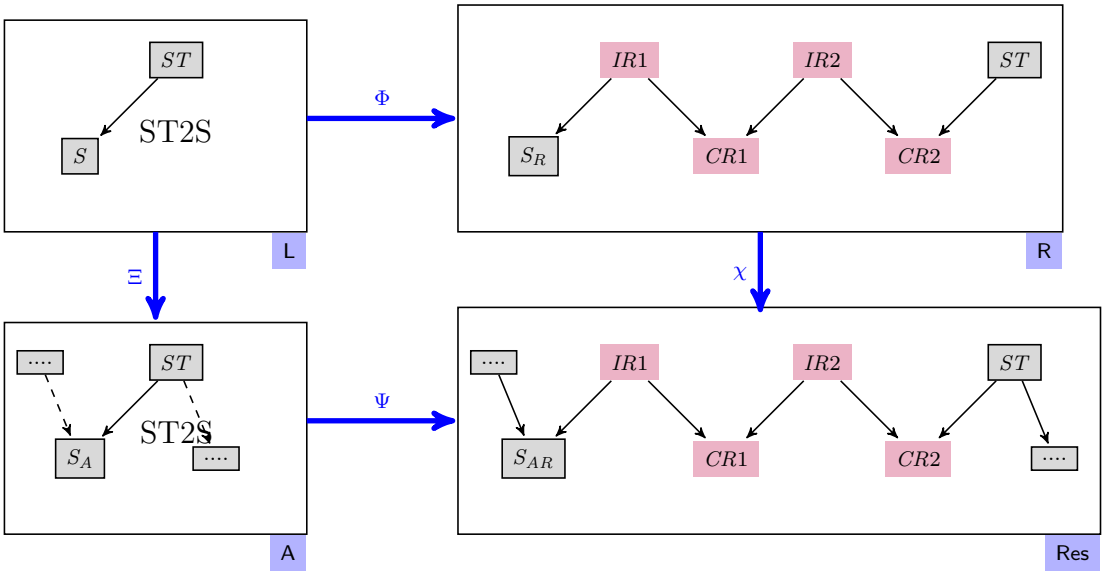


Figure 4.2.3: Component addition / component expansion

"Application" or the "Right Architecture" are directly copied. Similarly, connectors are also a direct copy of their preimage. For the above scenario in Figure 4.2.3, the component S_{AR} is a pushout construct of the cone $S_A \xleftarrow{\Xi} S \xrightarrow{\Phi} S_R$. Other components, i.e., CR_1, CR_2 are direct copies of the preimages from the "Right Architecture." All the connector parts, i.e., IR_1, IR_2, ST in the "Result Architecture" are also the direct copies of their preimages.

4.2.4 Component addition both ways:

Now, let us say, an edge is relaxed in both "Application" and "Right Architecture" by adding several 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 system architecture zigzag 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 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 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.

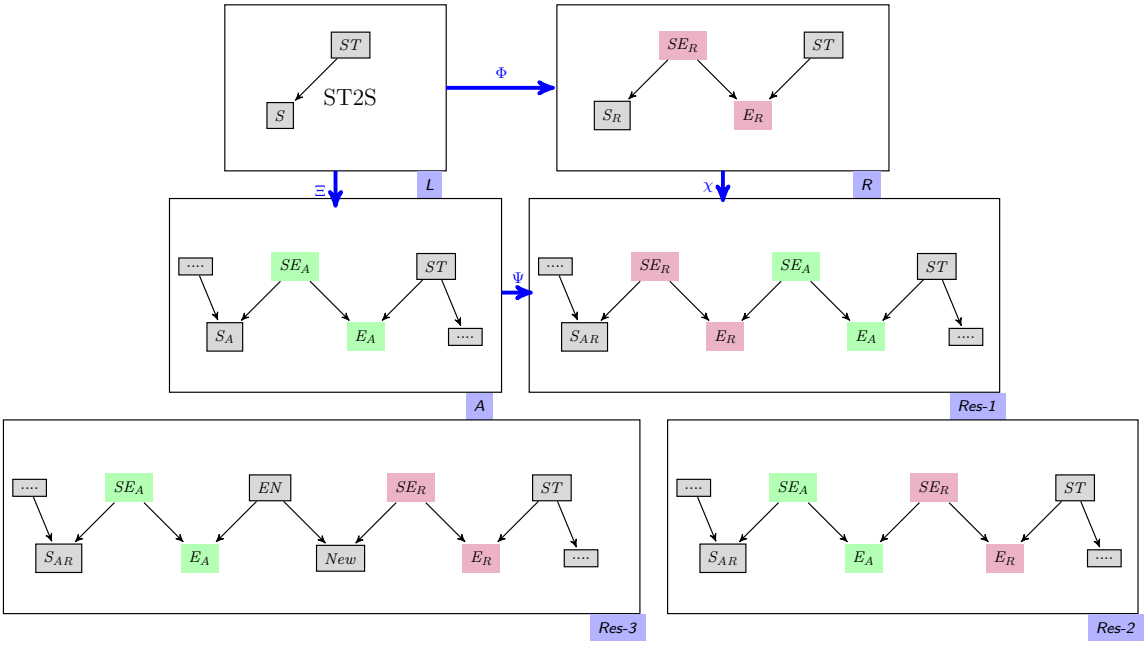


Figure 4.2.4: Component addition both ways

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 Architecture." A connection is established in the "Result Architecture" by breaking two connections; one each for the "Application" and the "Right Architecture." In the above example, for *Res*-1 the connection $SE_A \rightarrow S_A$ in the "Application Architecture" and $ST \rightarrow E_R$ in the "Right Architecture" got disconnected to establish a connection $SE_A to E_R$ in the "Result Architecture." This transformation is obvious if a specification homomorphism $SE_A \rightarrow ST$ exists. In other case, i.e., *Res*-2, the obvious transformation depends on the existence of the specification homomorphism $SE \rightarrow ST$. If neither of the homomorphisms exists, then we have to introduce new components and connectors to let the transformation take place and one of its kind is pictured by $Res - 3$ of the above figure.

In other words, component addition both ways case does not have any pushout in the category *SysArchsZ*. We have several potential pushout candidates with minimal components and connectors, but neither of them is a pushout object. There is system architecture zigzag homomorphism 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 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.

4.3 Construction of The Result Architecture: Restricted Pushout

In section 4.2, case by case we have described all the possibilities to construct a "Result Architecture" and we conjectured that in all cases except for the component addition both ways, the construction of the "Result Architecture", i.e., aspects introduction by zigzag graph transformations, is a pushout construct in the category *SysArchsZ*. In subsection 4.4, we prove all of those conjectures in 4.2.

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

4.3.1 SysArchsZ Restricted Pushout Theorem

Theorem 4.3.1 (Restricted pushout)**.** *Each span* $A \xleftarrow{\Phi} L \xrightarrow{\Xi} R$ *where no edge of* L *is mapped by both* Φ *and* Ξ *to non-trivial zigzag paths has a pushout.*

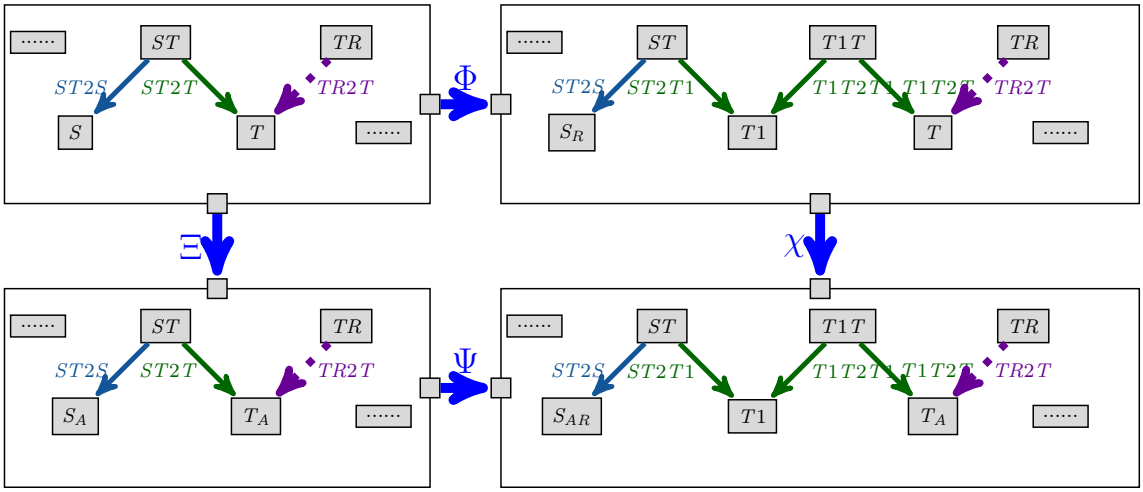


Figure 4.3.1: Sample Example: Restricted pushout

Proof: We have used induction principle to prove this theorem. Section 4.3.2 contains the proof of the induction principal, and section 4.4 and 4.3.3 respectively contains the proof of the basic steps.

4.3.2 Amalgamation Theorem

Although *SysArchsZ* 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 aspect 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 *SysArchsZ* pushouts can be amalgamated along *SysArchs* pushouts.

Theorem 4.3.2. *If the span* $\mathscr{A} \xleftarrow{\Xi} \mathscr{L} \xrightarrow{\Phi} \mathscr{R}$ *in SysArchsZ can be factored via three pushouts in SysArchs as shown in Figure 4.3.2, and if the two SysArchsZ spans* $\mathscr{A}_1 \xleftarrow{\Xi} \mathscr{L}_1 \xrightarrow{\Phi} \mathscr{R}_1$ *and* $\mathscr{A}_2 \xleftarrow{\Xi} \mathscr{L}_2 \xrightarrow{\Phi} \mathscr{R}_2$ *have pushouts in SysArchsZ, then* $\mathscr{A} \xleftarrow{\Xi} \mathscr{L} \xrightarrow{\Phi} \mathscr{R}$ *has pushout too.*

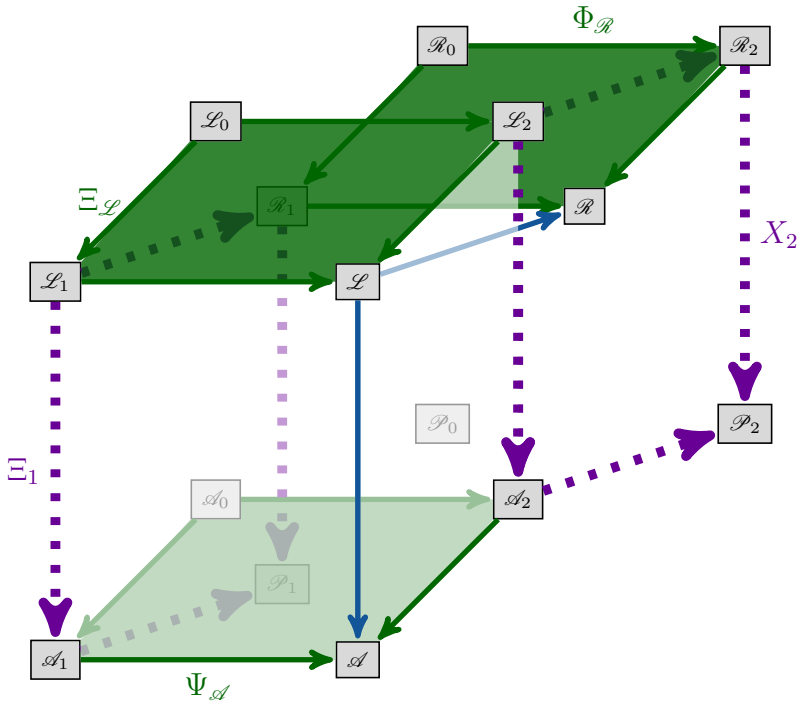


Figure 4.3.2: Amalgamation Theorem: Assumption

In the above Figure, the span $\mathscr{A} \xleftarrow{\Xi} \mathscr{L} \xrightarrow{\Phi} \mathscr{R}$ is factored via three pushouts in *SysArchs*, i.e., the $\mathscr{L}$-pushout, the $\mathscr{R}$-pushout and the $\mathscr{A}$-pushout. The $\mathscr{L}$-pushout is the pushout of the span $\mathscr{L}_1 \xleftarrow{\Xi_{\mathscr{L}}} \mathscr{L}_0 \xrightarrow{\Phi_{\mathscr{L}}} \mathscr{L}_2$, the $\mathscr{R}$-pushout is the pushout of the span $\mathscr{R}_1 \xleftarrow{\Xi_{\mathscr{R}}} \mathscr{R}_0 \xrightarrow{\Phi_{\mathscr{R}}} \mathscr{R}_2$, and the pushout of the span $\mathscr{A}_1 \xleftarrow{\Xi_{\mathscr{A}}} \mathscr{A}_0 \xrightarrow{\Phi_{\mathscr{A}}} \mathscr{A}_2$ is named as the $\mathscr{A}$-pushout. Two side pushouts are the 1-pushout and the 2-pushout, i.e., the pushout of the span $\mathscr{A}_1 \xleftarrow{\Xi_1} \mathscr{L}_1 \xrightarrow{\Phi_1} \mathscr{R}_1$ and $\mathscr{A}_2 \xleftarrow{\Xi_2} \mathscr{L}_2 \xrightarrow{\Phi_2} \mathscr{R}_2$ respectively.

The factoring makes the top square and the left-top square between the two cubes commute. They are represented in the following equation respectively.

$$\Phi_0 \ ; \ \Phi_{\mathscr{R}} = \Phi_{\mathscr{L}} \ ; \ \Phi_2 \text{ and } \Phi_0 \ ; \ \Xi_{\mathscr{R}} = \Xi_{\mathscr{L}} \ ; \ \Phi_1 \tag{1}$$

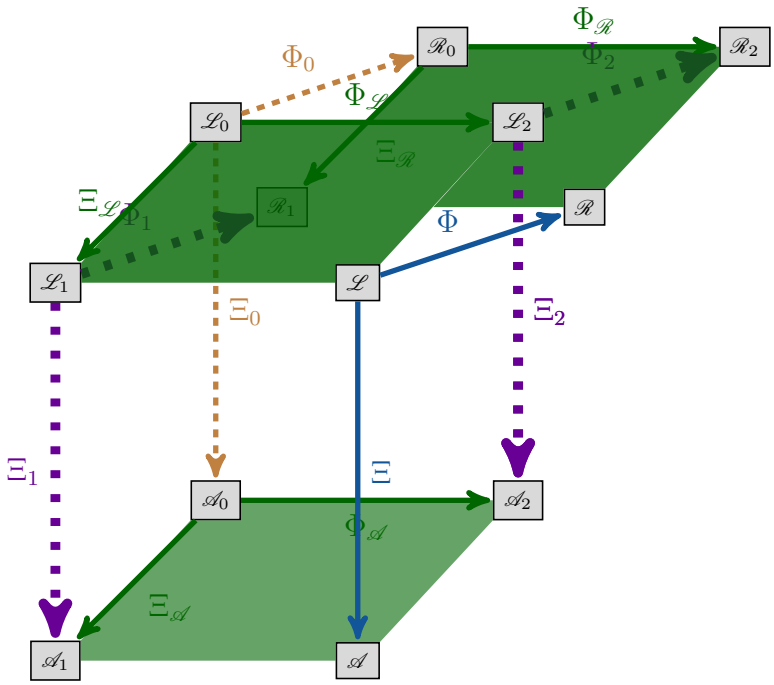


Figure 4.3.3: Amalgamation Theorem: Factoring

The factoring also makes the left-side square and the back square of the front cube commute. In equation:

$$\Xi_{\mathscr{L}} \ ; \ \Xi_1 = \Xi_0 \ ; \ \Xi_{\mathscr{A}} \text{ and } \Phi_{\mathscr{L}} \ ; \ \Xi_2 = \Xi_0 \ ; \ \Phi_{\mathscr{A}} \tag{2}$$

We need to prove that the span $\mathscr{A} \xleftarrow{\Xi} \mathscr{L} \xrightarrow{\Phi} \mathscr{R}$ has pushout. For a complete proof please see appendix A A.
This amalgamation theorem allows us to decompose prospective rule application pushouts of rules like those of Figure B.4.1 and Figure B.4.2 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 pieces in sections 4.3.3 and 4.4.

The situation is further simplified if we restrict ourselves to architectures with a *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 to be isomorphisms.

4.3.3 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 B.4.2, for which we show the corresponding rule fragment in Figure 4.3.4.

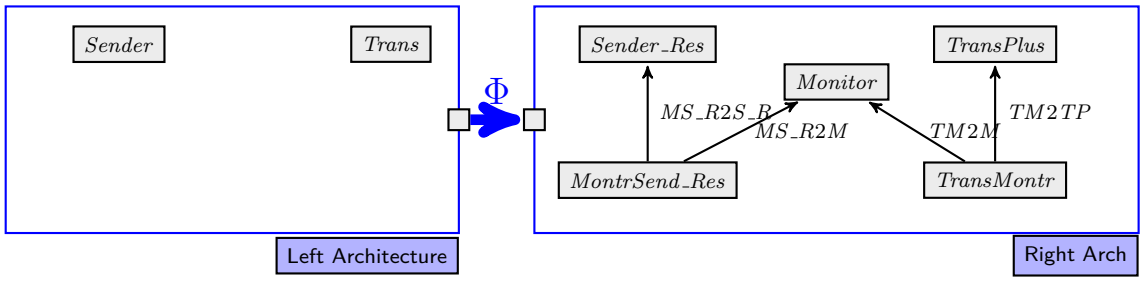


Figure 4.3.4: Discrete-LHS Fragment of Reliab. Intro.

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

Theorem 4.3.3. *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*.*

4.4 SysArchsZ Pushout Proofs for Single Edge LHS Cases

Theorem 4.4.1. *Pushout in* SysArchs *are preserved in* SysArchsZ *through* F_{SA}*, i.e., for evry span in* SysArchs*, a pushout in* SysArchs *for that span is also a pushout in* SysArchsZ*.*

Section 4.4.1 proves the theorem mentioned above.

4.4.1 Component expansion both ways:

The remainder of this section contains the proof structure of Theorem 4.4.1. If we restrict both rule and matching to non-zigzag homomorphisms, but allow the target of the "Left Architecture" 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 4.4.1. In the Figure 4.4.1 a single edge is extended both in "Application" and "Right Architecture," but for all of the *zigzag graph homomorphisms* $\Phi, \Xi, \Psi,$ and X the image of the *zigzag mapping* $\mathscr{H}_{ep}$ is a single edge *zigzag path.*

In Shape Graphs:

The shape graphs and morphisms between shape graphs constitute the *ZigzagGraphs* category where objects are graphs and morphism is defined as *zigzag graph homomorphism.*

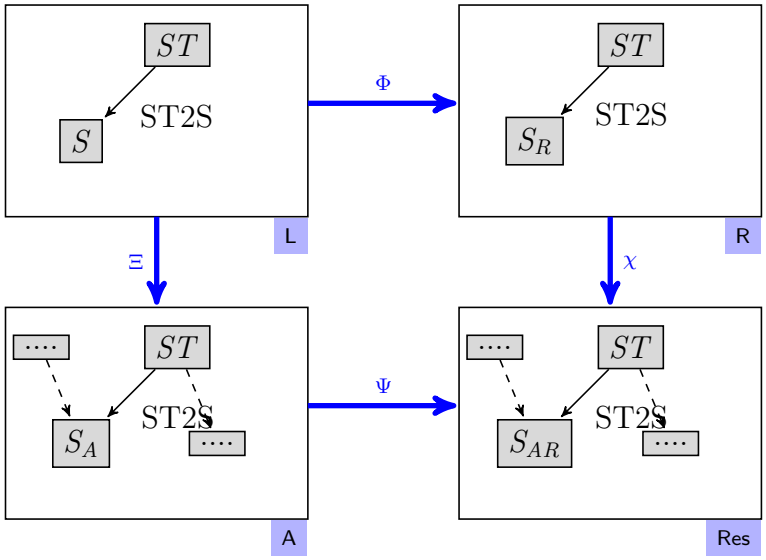


Figure 4.4.1: Component expansion both ways

Since the given span $\mathscr{A} \xleftarrow{\Xi} \mathscr{L} \xrightarrow{\Phi} \mathscr{R}$ in Figure 4.4.1 is a span in *Graphs* in Figure 4.4.1, the cospan $\mathscr{A} \xrightarrow{\Psi} \mathscr{P} \xleftarrow{X} \mathscr{R}$ is a pushout in *Graphs*. Hence, according to Theorem 3.7.1, this is a pushout in *ZigzagGraphs* category. So it is proved that, in this case *ZigzagGraphs* has a pushout, and the pushout object is the pushout object in *Graphs*.

In SysArchsZ:

System Architectures with system architecture homomorphisms, and system architecture zigzag homomorphisms constitute categories SysArchs and SysArchsZ respectively.

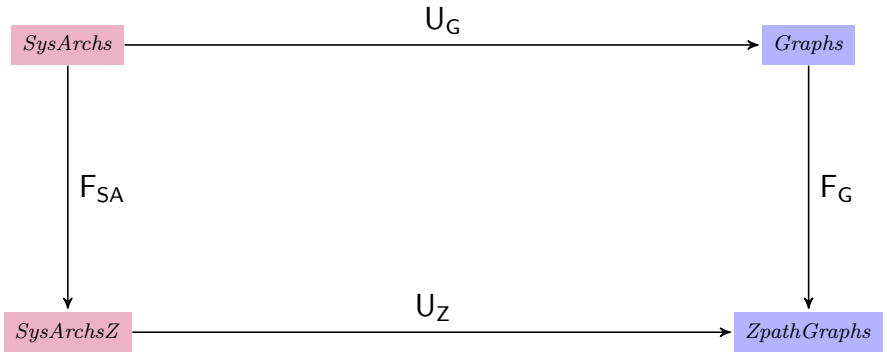


Figure 4.4.2: Pushout in SysArchsZ

According to the definition of *SysArchs* and *Graphs*, there is a forgetful functor $\mathsf{U_G}$: *SysArchs* → *Graphs* between them that forgets the $\mathscr{H}_{SpecMap}$ mapping and prop-

erties associated to it. There is another similar forgetful functor $\mathsf{U_Z} : \mathit{SysArchsZ} \rightarrow \mathit{ZpathGraphs}$ between *SysArchsZ* and *ZpathGraphs* that also forgets the $\mathscr{H}_{\mathit{SpecMap}}$ mapping and properties associated to it. According to theorem 3.7.1, functor $\mathsf{F_G}$ preserves the pushout of *Graphs* in *ZpathGraphs*. We need to show that, the functor $\mathsf{F_{SA}}$ also preserves the pushout of *SysArchs* in *SysArchsZ*.

First Step:

As a very first step we need to show that *SysArchs* has pushout.
In *SysArchs*, the pushout object constructed over $\Phi : \mathscr{L} \rightarrow \mathscr{R}$ and $\Xi : \mathscr{L} \rightarrow \mathscr{A}$ is the *system architecture* $\mathscr{P}$ along with the morphisms $\Psi : \mathscr{A} \rightarrow \mathscr{P}$ and $X : \mathscr{R} \rightarrow \mathscr{P}$ where,

1. The shape graph of $\mathscr{P}$ is the pushout object in **Graphs**.
2. System Architecture nodes (specifications) are constructed as pushouts in *DESC* or directly copied form Application and Right Architecture.
   (a) Nodes having preimages in Left Architecture $\mathscr{L}$ are constructed as pushouts.
   (b) Other nodes are constructed by directly copying them form architecture $\mathscr{A}$ and $\mathscr{R}$.
3. Edges (Specification Homomorphisms) in *system architecture* $\mathscr{P}$ are constructed as follows:
   (a) Edges having preimages in Left Architecture $\mathscr{L}$ will make the diagram in Figure 4.4.3 commutes, that is for all $e_p \in \mathscr{E}$ in $\mathscr{P}$ having preimages $e_l \in \mathscr{E}$ in $\mathscr{L}$ such that $(\mathscr{H}_\Phi(e_l), \mathscr{H}_\Xi(e_l)) = (e_r, e_a)$ will satisfy the conditions $\mathscr{R}(e_r)$; $\mathit{SpecMap}\ \mathit{trg}(e_r) = \mathscr{P}(e_p)$ and $\mathscr{A}(e_a)$; $\mathit{SpecMap}\ \mathit{trg}(e_a) = \mathscr{P}(e_p)$.
   (b) Other edges will have preimages either in Application Architecture $\mathscr{A}$ or Right Architecture $\mathscr{R}$. Those edges are constructed such that the diagram associated to the edge and it's preimage will commutes.
      (i) For all $e_p \in \mathscr{E}$ in $\mathscr{P}$ having preimages $e_a \in \mathscr{E}$ only in $\mathscr{A}$ such that $\mathscr{A}(\mathit{trg}(e_a)) \neq \mathscr{P}(\mathscr{H}_\mathscr{Y}\, \mathit{trg}(e_a))$ will satisfy the conditions $\mathscr{A}(e_a)$; $\mathit{SpecMap}\ \mathit{trg}(e_a) = \mathscr{P}(e_p)$
      (ii) For all $e_p \in \mathscr{E}$ in $\mathscr{P}$ having preimages $e_a \in \mathscr{E}$ only in $\mathscr{A}$ such that $\mathscr{A}(\mathit{trg}(e_a)) = \mathscr{P}(\mathscr{H}_\mathscr{Y}\, \mathit{trg}(e_a))$ and $\mathscr{H}_{\mathit{SpecMap}}\ \mathit{trg}(e_a) = \mathit{SpecId}\ \mathscr{A}_1\ \mathit{trg}(e_a)$, the edges would be a direct copy of it's preimages.
      (iii) Edges having preimages only in Right Architecture $\mathscr{R}$ will be constructed similarly following (i), and (ii).

For any other given System Architecture $\mathscr{P}'$ and morphisms $\Psi' : \mathscr{A} \rightarrow \mathscr{P}'$ and $X' : \mathscr{R} \rightarrow \mathscr{P}'$ with $\Phi;X' = \Xi;\Psi'$ we need to prove that there is a unique morphism $\mathscr{U} : \mathscr{P} \rightarrow \mathscr{P}'$ such that $X;\mathscr{U} = X'$ and $\Psi;\mathscr{U} = \Psi'$.

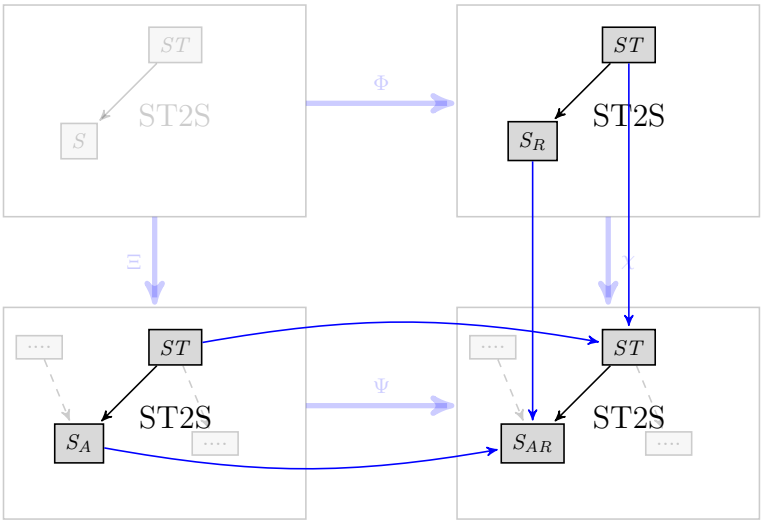


Figure 4.4.3: Diagram in *DESC*

Category *Graphs* has pushouts, therefore a unique homomorphism $\mathscr{U} : \mathscr{P} \to \mathscr{P}'$ exists at the shape graph level. Now we need to show that this homomorphism can be extended to a *system architecture homomorphism*.
The *system architecture* nodes for architecture $\mathscr{P}$ and $\mathscr{P}'$ are constructed as pushouts in *DESC*. So, the morphism $\mathscr{U} : \mathscr{P} \to \mathscr{P}'$ satisfies the *IdentityMap* and *Commutes* property defined in 3.7.2. Hence, the morphism $\mathscr{U}$ is a *system architecture homomorphism*.
So, it is shown that *SysArchs* category has pushout.

Final Step:

Now we need to show that, pushout in *SysArchs* are preserved in *SysArchsZ* through $\mathsf{F_{SA}}$.

According to theorem 3.7.1, a pushout of a graphs span in *Graphs* is a *ZigzagGraphs* pushout. That is in Figure 4.4.2, the functor $\mathsf{F_G}$ preserves the pushout in *Graphs*. For given a pushout in *SysArchs* and another candidate in *SysArchsZ*, according to Figure 4.4.2, there is a unique *system architecture homomorphism* between pushout object and the other pushout candidate. Now, we need to show that this *system architecture homomorphism* can be extended to a *system architecture zigzag graph homomorphism*. Since *system architecture zigzag graph homomorphism* does not impose any commutativity, *system architecture homomorphism* can directly be extended to the *system architecture zigzag graph homomorphism*.
So, it is proved that pushout in *SysArchs* are preserved in *SysArchsZ* through $\mathsf{F_{SA}}$.

Theorem 4.4.2. *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.

4.4.2 Component addition / component expansion:

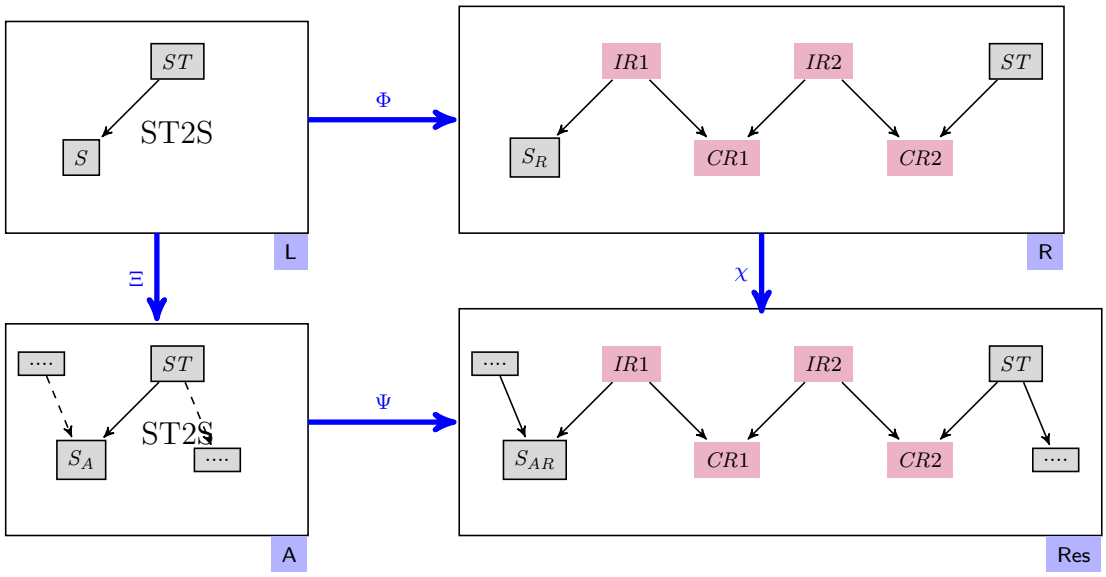


Figure 4.4.4: Component addition / component expansion

In the *Component addition / component expansion* cases, a singe edge is relaxed either in "Application" or "Right Architecture", but extended or remain unchanged (exact matched) in the opposite/other architecture. According to the definition, *zigzag graph homomorphism* maps a single edge to a *zigzag path*, but in this case either Φ or Ξ will map a single edge to a *zigzag path* that is actually a single edge. In the above example, the "Right Architecture" get relaxed and the "Application Architecture" get extended.

In Shape Graphs:

The shape graphs and morphisms between shape graphs constitute the *ZigzagGraphs* category where *objects* are graphs and *homomorphism* is defined as *zigzag graph homomorphism.*

For a given span $\mathscr{A} \xleftarrow{\Xi} \mathscr{L} \xrightarrow{\Phi} \mathscr{R}$ in *ZigzagGraphs* category, where the image of the $\mathscr{H}_{ep}$ mapping in Ξ is a single edge, we need to prove that the cospan $\mathscr{A} \xrightarrow{\Psi} \mathscr{P} \xleftarrow{X} \mathscr{R}$ is a pushout. In order to prove this, we need to show that the cospan is universal, that is for any other cospan $\mathscr{A} \xrightarrow{\Psi'} \mathscr{P}' \xleftarrow{X'} \mathscr{R}$ for which the extension of the above diagram commutes, there must exist a unique $\mathscr{U} : \mathscr{P} \to \mathscr{P}'$ also making the diagram commutes.

For any other given cocone $\mathscr{A} \xrightarrow{\Psi'} \mathscr{P}' \xleftarrow{X'} \mathscr{R}$ where $\Phi ; X' = \Xi ; \Psi'$, there is a $\mathscr{R}$-$\mathscr{Z}path$ graph morphism $\mathscr{U}_R : \mathscr{P} \leftrightarrow \mathscr{P}'$ such that $X ; \mathscr{U}_R = X'$and,$\Psi ; \mathscr{U}_R = \Psi'$. The $\mathscr{R}$-$\mathscr{Z}path$ graph morphism is defined as follows:

$$\mathscr{U}_R = (\mathscr{U}_{R,\mathscr{V}} \ , \quad \iota ; \mathscr{U}_{R,\mathscr{E}})$$

Where

$$\mathscr{U}_{R,\mathscr{V}} = (\Psi_{\mathscr{V}}^{\smile} ; \Psi'_{\mathscr{V}} \cup X_{\mathscr{V}}^{\smile} ; X'_{\mathscr{V}}) \text{ and } \mathscr{U}_{R,\mathscr{E}} = ((\Psi_{\mathscr{E}})^{\smile} ; \Psi'_{\mathscr{E}} \cup (X_{\mathscr{E}})^{\smile} ; X'_{\mathscr{E}})$$

We need to show that $\iota ; \mathscr{U}_{R,\mathscr{E}}$ is univalent and total.

Univalent:

If $\iota ; \mathscr{U}_{R,\mathscr{E}}$ is an univalent relation, then it implies $(\iota ; \mathscr{U}_{R,\mathscr{E}})^{\smile} ; (\iota ; \mathscr{U}_{R,\mathscr{E}}) \subseteq \mathbb{I}$. The proof is given as follows:

L.H.S

$=$ $\langle = \rangle$

$(\iota ; \mathscr{U}_{R,\,\mathscr{E}})^{\smile} ; (\iota ; \mathscr{U}_{R,\,\mathscr{E}})$

$=$ $\langle$Substitute $\mathscr{U}_{R,\mathscr{E}}\rangle$

$(\iota ; ((\Psi_{\mathscr{E}})^{\smile} ; \Psi'_{\mathscr{E}} \cup (X_{\mathscr{E}})^{\smile} ; X'_{\mathscr{E}}))^{\smile} ; (\iota ; ((\Psi_{\mathscr{E}})^{\smile} ; \Psi'_{\mathscr{E}} \cup (X_{\mathscr{E}})^{\smile} ; X'_{\mathscr{E}}))$

$=$ $\langle$distribute ; over $\cup\rangle$

$(\iota ; (\Psi_{\mathscr{E}})^{\smile} ; \Psi'_{\mathscr{E}} \cup \iota ; (X_{\mathscr{E}})^{\smile} ; X'_{\mathscr{E}})^{\smile} ; (\iota ; (\Psi_{\mathscr{E}})^{\smile} ; \Psi'_{\mathscr{E}} \cup \iota ; (X_{\mathscr{E}})^{\smile} ; X'_{\mathscr{E}})$

$=$ $\left\langle \text{Apply } {}^{\smile} \right\rangle$

$((\Psi'_{\mathscr{E}})^{\smile} ; \Psi_{\mathscr{E}} ; \iota^{\smile} \cup (X'_{\mathscr{E}})^{\smile} ; X_{\mathscr{E}} ; \iota^{\smile}) ; (\iota ; (\Psi_{\mathscr{E}})^{\smile} ; \Psi'_{\mathscr{E}} \cup \iota ; (X_{\mathscr{E}})^{\smile} ; X'_{\mathscr{E}})$

$=$ $\langle$Distribute ; over $\cup\rangle$

$(\Psi'_{\mathscr{E}})^{\smile} ; \Psi_{\mathscr{E}} ; \iota^{\smile} ; \iota ; (\Psi_{\mathscr{E}})^{\smile} ; \Psi'_{\mathscr{E}} \cup (\Psi'_{\mathscr{E}})^{\smile} ; \Psi_{\mathscr{E}} ; \iota^{\smile} ; \iota ; (X_{\mathscr{E}})^{\smile} ; X'_{\mathscr{E}} \cup$
$(X'_{\mathscr{E}})^{\smile} ; X_{\mathscr{E}} ; \iota^{\smile} ; \iota ; (\Psi_{\mathscr{E}})^{\smile} ; \Psi'_{\mathscr{E}} \cup (X'_{\mathscr{E}})^{\smile} ; X_{\mathscr{E}} ; \iota^{\smile} ; \iota ; (X_{\mathscr{E}})^{\smile} ; X'_{\mathscr{E}}$

$$
\begin{array}{ll}
= & \Big\langle \Psi_{\mathscr{E}} \,;\, \iota^{\smile} \,;\, \iota \,;\, (X_{\mathscr{E}})^{\smile} = \varnothing,\ X_{\mathscr{E}} \,;\, \iota^{\smile} \,;\, \iota \,;\, (\Psi_{\mathscr{E}})^{\smile} = \varnothing \Big\rangle \\
& (\Psi'_{\mathscr{E}})^{\smile} \,;\, \Psi_{\mathscr{E}} \,;\, \iota^{\smile} \,;\, \iota \,;\, (\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi'_{\mathscr{E}} \cup (X'_{\mathscr{E}})^{\smile} \,;\, X_{\mathscr{E}} \,;\, \iota^{\smile} \,;\, \iota \,;\, (X_{\mathscr{E}})^{\smile} \,;\, X'_{\mathscr{E}} \\
\subseteq & \Big\langle \iota^{\smile} \,;\, \iota \subseteq \mathbb{I}, \text{ Identity Law} \Big\rangle \\
& (\Psi'_{\mathscr{E}})^{\smile} \,;\, \Psi_{\mathscr{E}} \,;\, (\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi'_{\mathscr{E}} \cup (X'_{\mathscr{E}})^{\smile} \,;\, X_{\mathscr{E}} \,;\, (X_{\mathscr{E}})^{\smile} \,;\, X'_{\mathscr{E}} \\
\subseteq & \langle \Psi_{\mathscr{E}}, X_{\mathscr{E}} \text{ are Injective, Identity Law} \rangle \\
& (\Psi'_{\mathscr{E}})^{\smile} \,;\, \Psi'_{\mathscr{E}} \cup (X'_{\mathscr{E}})^{\smile} \,;\, X'_{\mathscr{E}} \\
\subseteq & \langle \Psi'_{\mathscr{E}} \text{ and } X'_{\mathscr{E}} \text{ are univalent} \rangle \\
& \mathbb{I}
\end{array}
$$

$\Psi_{\mathscr{E}}$ maps the edge $ST \rightarrow S_A$ to the *Zpath* $S_{AR} \leftarrow IR_1 \rightarrow ..... \leftarrow ST$. According to the definition of Theorem 3.5.2 and relation composition, $\Psi_{\mathscr{E}} \,;\, \iota^{\smile} = \varnothing$. Hence, $\Psi_{\mathscr{E}} \,;\, \iota^{\smile} \,;\, \iota \,;\, (X_{\mathscr{E}})^{\smile} = \varnothing \,;\, \iota \,;\, (X_{\mathscr{E}})^{\smile} = \varnothing$. From the definition of $X_{\mathscr{E}}$ and $(\Psi_{\mathscr{E}})^{\smile}$ in the Figure 4.4.4, it is clear that $ran(X_{\mathscr{E}};\iota^{\smile};\iota) \cap dom((\Psi_{\mathscr{E}})^{\smile}) = \varnothing$. Hence, $X_{\mathscr{E}};\iota^{\smile};\iota;(\Psi_{\mathscr{E}})^{\smile} = \varnothing$.

Total:

If $\iota \,;\, \mathscr{U}_{R,\mathscr{E}}$ is a total relation, then it implies $\mathbb{I} \subseteq \iota \,;\, \mathscr{U}_{R,\mathscr{E}} \,;\, (\iota \,;\, \mathscr{U}_{R,\mathscr{E}})^{\smile}$. The proof is scratched as follows:

$$
\begin{array}{ll}
 & \textbf{R.H.S} \\
= & \langle\rangle \\
& (\iota \,;\, \mathscr{U}_{R,\,\mathscr{E}}) \,;\, (\iota \,;\, \mathscr{U}_{R,\,\mathscr{E}})^{\smile} \\
= & \langle \text{Substitute } \mathscr{U}_{R,\mathscr{E}} \rangle \\
& (\iota \,;\, ((\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi'_{\mathscr{E}} \cup (X_{\mathscr{E}})^{\smile} \,;\, X'_{\mathscr{E}})) \,;\, (\iota \,;\, ((\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi'_{\mathscr{E}} \cup (X_{\mathscr{E}})^{\smile} \,;\, X'_{\mathscr{E}}))^{\smile} \\
= & \langle \text{distribute ; over } \cup \rangle \\
& (\iota \,;\, (\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi'_{\mathscr{E}} \cup \iota \,;\, (X_{\mathscr{E}})^{\smile} \,;\, X'_{\mathscr{E}}) \,;\, (\iota \,;\, (\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi'_{\mathscr{E}} \cup \iota \,;\, (X_{\mathscr{E}})^{\smile} \,;\, X'_{\mathscr{E}})^{\smile} \\
= & \Big\langle \text{Apply } {}^{\smile} \Big\rangle \\
& (\iota \,;\, (\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi'_{\mathscr{E}} \cup \iota \,;\, (X_{\mathscr{E}})^{\smile} \,;\, X'_{\mathscr{E}}) \,;\, ((\Psi'_{\mathscr{E}})^{\smile} \,;\, \Psi_{\mathscr{E}} \,;\, \iota^{\smile} \cup (X'_{\mathscr{E}})^{\smile} \,;\, X_{\mathscr{E}} \,;\, \iota^{\smile}) \\
= & \langle \text{Distribute ; over } \cup \rangle \\
& \iota \,;\, (\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi'_{\mathscr{E}} \,;\, (\Psi'_{\mathscr{E}})^{\smile} \,;\, \Psi_{\mathscr{E}} \,;\, \iota^{\smile} \cup \iota \,;\, (\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi'_{\mathscr{E}} \,;\, (X'_{\mathscr{E}})^{\smile} \,;\, X_{\mathscr{E}} \,;\, \iota^{\smile} \cup \\
& \iota \,;\, (X_{\mathscr{E}})^{\smile} \,;\, X'_{\mathscr{E}} \,;\, (\Psi'_{\mathscr{E}})^{\smile} \,;\, \Psi_{\mathscr{E}} \,;\, \iota^{\smile} \cup \iota \,;\, (X_{\mathscr{E}})^{\smile} \,;\, X'_{\mathscr{E}} \,;\, (X'_{\mathscr{E}})^{\smile} \,;\, X_{\mathscr{E}} \,;\, \iota^{\smile} \\
\supseteq & \langle \Psi'_{\mathscr{E}}, X'_{\mathscr{E}} \text{ are total;} \rangle \\
& \iota \,;\, (\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi_{\mathscr{E}} \,;\, \iota^{\smile} \cup \iota \,;\, (X_{\mathscr{E}})^{\smile} \,;\, X_{\mathscr{E}} \,;\, \iota^{\smile} \\
\supseteq & \langle \text{Distributive Law} \rangle \\
& \iota \,;\, ((\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi_{\mathscr{E}} \,;\, \cup (X_{\mathscr{E}})^{\smile} \,;\, X_{\mathscr{E}}) \,;\, \iota^{\smile}
\end{array}
$$

$\supseteq$ $\langle \Psi$ and X are univalent and jointly surjective$\rangle$
$\iota \,;\, \mathbb{I} \,;\, \iota^{\smile}$
$\supseteq$ $\langle$Definition of $\iota\rangle$
$\mathbb{I}$

Uniqueness:

Assume, there is another $\mathscr{R}$-$\mathscr{Z}path$ graph homomorphism $\mathscr{W}_{\mathscr{R}} : \mathscr{P} \leftrightarrow \mathscr{P}'$ such that $X_{\mathscr{V}} \,;\, \mathscr{W}_{\mathscr{R},\mathscr{V}} = X'_{\mathscr{V}}$, $\Psi_{\mathscr{V}} \,;\, \mathscr{W}_{\mathscr{R},\mathscr{V}} = \Psi'_{\mathscr{V}}$ and $X_{\mathscr{E}} \,;\, \mathscr{W}_{\mathscr{R},\mathscr{E}} = X'_{\mathscr{E}}$ and,$\Psi_{\mathscr{E}} \,;\, \mathscr{W}_{\mathscr{R},\mathscr{E}} = \Psi'_{\mathscr{E}}$. Now we need to show that $\mathscr{U}_{\mathscr{R}}$ is unique. That is:

$\mathscr{U}_{R,\,\mathscr{V}} = \mathscr{W}_{R,\,\mathscr{V}}$ and $\mathscr{U}_{R,\,\mathscr{E}} = \mathscr{W}_{R,\,\mathscr{E}}$.

$\mathscr{U}_{R,\,\mathscr{V}}$
$=$ $\langle$Definition$\rangle$
$(\Psi_{\mathscr{V}})^{\smile}$ **;** $\Psi'_{\mathscr{V}} \cup (X_{\mathscr{V}})^{\smile}$ **;** $X'_{\mathscr{V}}$
$=$ $\langle$Substitute $X'_{\mathscr{V}}$ and $\Psi'_{\mathscr{V}}\rangle$
$(\Psi_{\mathscr{V}})^{\smile}$ **;** $\Psi_{\mathscr{V}}$ **;** $\mathscr{W}_{\mathscr{R},\mathscr{V}} \cup (X_{\mathscr{V}})^{\smile}$ **;** $X_{\mathscr{V}}$ **;** $\mathscr{W}_{\mathscr{R},\mathscr{V}}$
$=$ $\langle$Distributive Law$\rangle$
$(\Psi_{\mathscr{V}}{}^{\smile} \,;\, \Psi_{\mathscr{V}} \cup X_{\mathscr{V}}{}^{\smile} \,;\, X_{\mathscr{V}}) \,;\, \mathscr{W}_{\mathscr{R},\mathscr{V}}$
$=$ $\langle \Psi_{\mathscr{V}}$ and $X_{\mathscr{V}}$ are univalent and jointly surjective$\rangle$
$\mathbb{I} \,;\, \mathscr{W}_{\mathscr{R},\mathscr{V}}$
$=$ $\langle$Identity Law$\rangle$
$\mathscr{W}_{\mathscr{R},\mathscr{V}}$

$\mathscr{U}_{R,\,\mathscr{E}}$
$=$ $\langle$Definition$\rangle$
$(\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi'_{\mathscr{E}} \cup (X_{\mathscr{E}})^{\smile} \,;\, X'_{\mathscr{E}}$
$=$ $\langle$Substitute $X'_{\mathscr{E}}$ and $\Psi'_{\mathscr{E}}\rangle$
$(\Psi_{\mathscr{E}})^{\smile} \,;\, \Psi_{\mathscr{E}} \,;\, \mathscr{W}_{\mathscr{R},\mathscr{E}} \cup (X_{\mathscr{E}})^{\smile} \,;\, X_{\mathscr{E}} \,;\, \mathscr{W}_{\mathscr{R},\mathscr{E}}$
$=$ $\langle$Distributive Law$\rangle$
$(\Psi_{\mathscr{E}}{}^{\smile} \,;\, \Psi_{\mathscr{E}} \cup X_{\mathscr{E}}{}^{\smile} \,;\, X_{\mathscr{E}}) \,;\, \mathscr{W}_{\mathscr{R},\mathscr{E}}$
$=$ $\langle \Psi_{\mathscr{E}}$ and $X_{\mathscr{E}}$ are univalent and jointly surjective$\rangle$
$\mathbb{I} \,;\, \mathscr{W}_{\mathscr{R},\mathscr{E}}$
$=$ $\langle$Identity Law$\rangle$
$\mathscr{W}_{\mathscr{R},\mathscr{E}}$

So, it is shown that $\mathscr{U}_R$ is unique.

$\mathscr{U}_R$ is a *zigzag path*:

In order to show that $\mathscr{R}$-$\mathscr{Z}$*path* graph homomorphism $\mathscr{U}_R$ is a *zigzag graph homomorphism* $\mathscr{U}$, we need to show that $\mathscr{U}_R = (\mathscr{U}_{R,\mathscr{V}} \ , \quad \iota \,;\mathscr{U}_{R,\mathscr{E}})$, i.e., $\mathscr{U}_{R,\mathscr{V}} \wedge \iota \,;\mathscr{U}_{R,\mathscr{E}}$ is total and univalent. In this section we have proved that the second part, i.e., $\iota \,;\mathscr{U}_{R,\mathscr{E}}$ is total and univalent and the proof for the first part, i.e., $\mathscr{U}_{R,\mathscr{V}}$ is shown in Theorem 3.7.1.
So, it is proved that the above diagram is a pushout in *ZigzagGraphs*.

In SysArchsZ:
The *system architectures* and *system architecture zigzag homomorphism* constitute category *SysArchsZ*. We need to show that for this *SingleRelax* case, the *SysArchsZ* has a pushout.

In *SysArchsZ*, the pushout object constructed over the morphisms $\Phi : \mathscr{L} \to \mathscr{R}$ and $\Xi : \mathscr{L} \to \mathscr{A}$ in Figure 4.4.4 is the *system architecture* $\mathscr{P}$ along with the morphisms $\Psi : \mathscr{A} \to \mathscr{P}$ and $X : \mathscr{R} \to \mathscr{P}$ where,

1. The shape graph of $\mathscr{P}$ is the pushout object in *ZpathGraphs*.
2. *system architecture* nodes (specifications) and edges (Specification Homomorphisms) will be constructed by following the procedure described in item **2.**, and item **3.** of Section 4.4.1.

For any other given *system architecture* $\mathscr{P}'$ and morphisms $\Psi' : \mathscr{A} \to \mathscr{P}'$ and $X' : \mathscr{R} \to \mathscr{P}'$ with $\Phi;X' = \Xi;\Psi'$ we need to prove that there is a unique morphism $\mathscr{U} : \mathscr{P} \to \mathscr{P}'$ such that $X;\mathscr{U} = X'$ and $\Psi;\mathscr{U} = \Psi'$.
In *SingleRelax* case, the category *ZpathGraphs* has pushout, therefore a unique homomorphism $\mathscr{U} : \mathscr{P} \to \mathscr{P}'$ exists at the shape graph level. Now we need to show that this homomorphism can be extended to a *system architecture zigzag homomorphism*. The *system architecture* nodes for architecture $\mathscr{P}$ and $\mathscr{P}'$ are constructed as pushouts in *DESC*. In Section 4.4.1 it is shown that pushout in *SysArchs* are preserved in *SysArchsZ*. So, the morphism $\mathscr{U} : \mathscr{P} \to \mathscr{P}'$ satisfies the *IdentityMap*, *Complete* and *Empty* property defined in Section 3.7.3. Hence, the morphism $\mathscr{U}$ is a *system architecture zigzag homomorphism*.

So, it is shown that for *SingleRelax* case, the category *SysArchsZ* has pushout.

4.4.3 Component addition / identity matching:

The *Component addition / identity matching* case, where we find an exact match in *application architecture* and a *zigzag match* in *right architecture*, is a simple version of the *Component addition / component expansion* case proved in Section 4.4.2. 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 4.4.5.

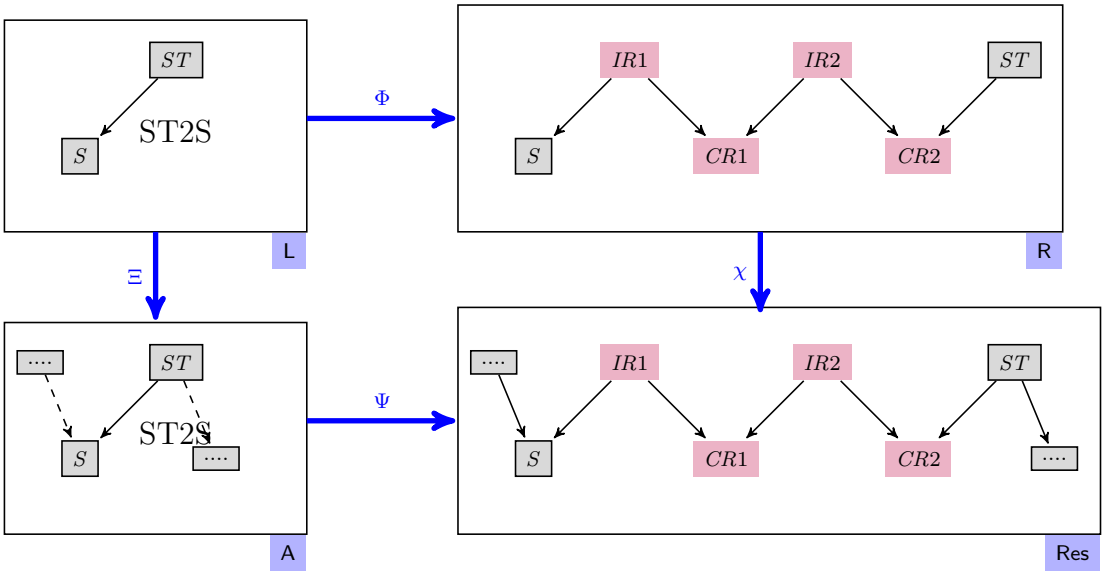


Figure 4.4.5: Component addition / identity matching

Theorem 4.4.3. *For* "Component addition both ways" *case spans, the* SysArchsZ *category does not have any pushouts.*

4.5 Component addition both ways:

In the *Component addition both ways* cases, a singe edge is relaxed both in "Application" and "Right Architecture". If we take the span in the Figure 4.2.4 as an example, there must be a path from ST to S_{AR} containing images of SE_A,E_A,SE_R,E_R in the pushout object. These four images could be four individual nodes or less than four nodes.

Four Nodes:

If we follow the pushout construct procedure by considering four individual images, we come up with several potential pushout objects, e.g., A, and R, depicted in the Figure 4.5.1. The Figure 4.5.1 also contains two pushout candidates which make the outer square of the Figure 4.2.4 commute.

For the given span in Figure 4.2.4, if we consider *Pushout object A* as a pushout object and *Candidate R* as a pushout candidate we will come up with the following situations where:

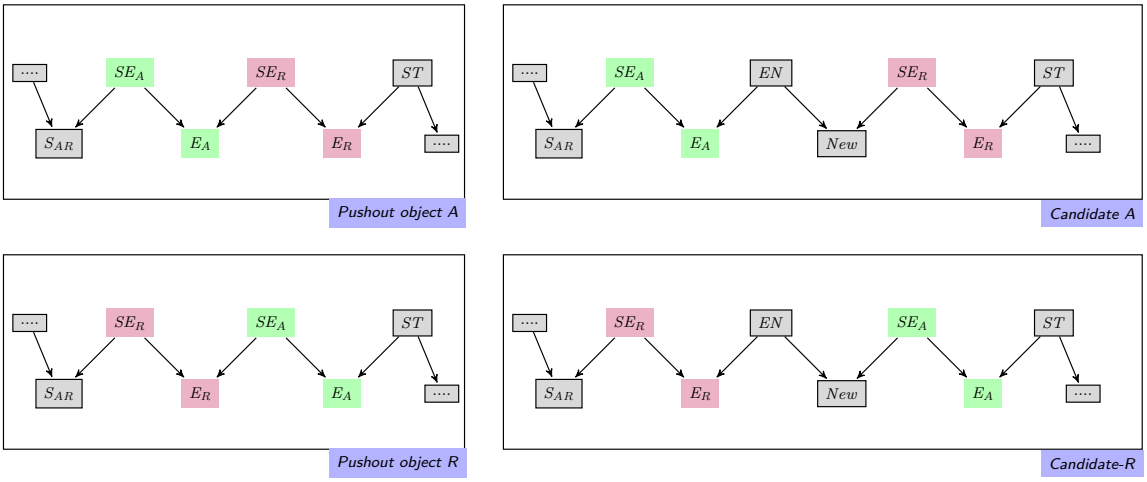


Figure 4.5.1: Pushout objects and pushout candidates for Figure 4.2.4

$\underline{\Phi\text{: } \boldsymbol{L \mapsto R}}$	$\underline{\Xi\text{: } \boldsymbol{L \mapsto A}}$
S $\leftarrow$ ST $\mapsto$ S_R $\leftarrow$ $SE_R \rightarrow E_R \leftarrow ST$	S $\leftarrow$ ST $\mapsto$ $S_A \leftarrow SE_A \rightarrow E_A \leftarrow ST$
$\underline{\Psi\text{: } \boldsymbol{A \mapsto PO\text{-}A}}$	$\underline{X\text{: } \boldsymbol{R \mapsto PO\text{-}A}}$
$S_A \leftarrow SE_A \mapsto S_{AR} \leftarrow SE_A$ $SE_A \rightarrow E_A \mapsto SE_A \rightarrow E_A$ $E_A \leftarrow ST \mapsto E_A \leftarrow SE_R \rightarrow E_R \leftarrow ST$	$S_R \leftarrow SE_R \mapsto S_{AR} \leftarrow SE_A \rightarrow E_A \leftarrow SE_R$ $SE_R \rightarrow E_R \mapsto SE_R \rightarrow E_R$ $E_R \leftarrow ST \mapsto E_R \leftarrow ST$
$\underline{\Psi_A\text{: } \boldsymbol{A \mapsto CAN\text{-}R}}$	$\underline{X_A\text{: } \boldsymbol{R \mapsto CAN\text{-}R}}$
$S_A \leftarrow SE_A \mapsto S_{AR} \leftarrow SE_R \rightarrow E_R \leftarrow EN \rightarrow New \leftarrow SE_A$ $SE_A \rightarrow E_A \mapsto SE_A \rightarrow E_A$ $E_A \leftarrow ST \mapsto E_A \leftarrow ST$	$S_R \leftarrow SE_R \mapsto S_{AR} \leftarrow SE_R$ $SE_R \rightarrow E_R \mapsto SE_R \rightarrow E_R$ $E_R \leftarrow ST \mapsto E_R \leftarrow EN \rightarrow New \leftarrow SE_A \rightarrow E_A \leftarrow ST$

$\underline{U_A\text{: } \boldsymbol{PO\text{-}A \mapsto CAN\text{-}R}}$

$S_{AR} \leftarrow SE_A \mapsto S_{AR} \leftarrow SE_R \rightarrow E_R \leftarrow EN \rightarrow New \leftarrow SE_A,$
$SE_A \rightarrow E_A \mapsto SE_A \rightarrow E_A,$
$E_A \leftarrow SE_R \mapsto E_A \leftarrow SE_A \rightarrow New \leftarrow EN \rightarrow E_R \leftarrow SE_R,$
$SE_R \rightarrow E_R \mapsto SE_R \rightarrow E_R,$
$E_R \leftarrow ST \mapsto E_R \leftarrow EN \rightarrow New \leftarrow SE_A \rightarrow E_A \leftarrow ST$

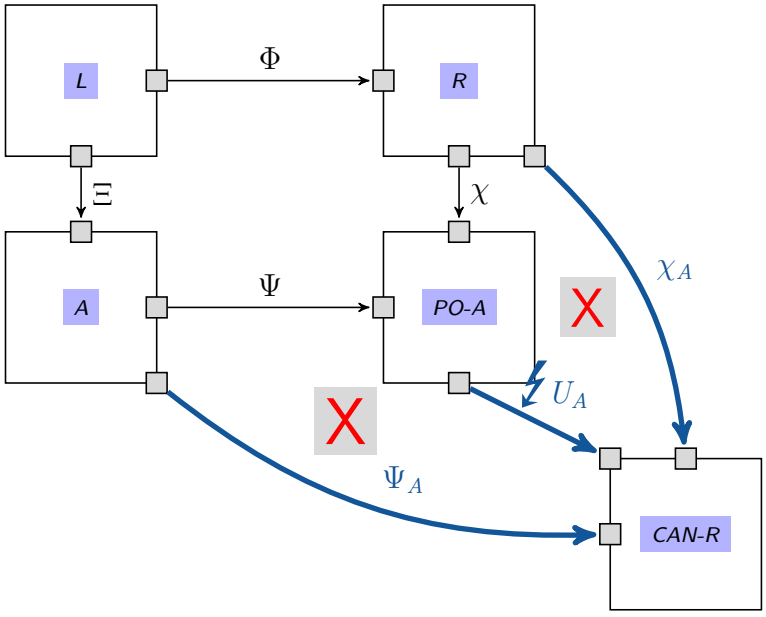


Figure 4.5.2: *AllRelax* case is *SysArchsZ* has no pushout

Here, $\Phi;X_A = \Xi;\Phi_A$, U_A is the unique morphism but $X;U_A \neq X_A$ and $\Phi;U_A \neq \Phi_A$. If we consider "Pushout object" R as a *pushout object* and "Candidate A" as a *pushout candidate* we come up with a similar situation where the outer square commutes, there is a *unique morphism* between the "pushout object" and the "candidate object" but neither of the inner square commute. In another word, there is no morphism between "PO-A" and "CAN-R" that makes all the squares in Figure 4.5.2 commute. So, it is proved that *Component addition both ways* case does not have pushout.

Less Than Four Nodes:

If we consider pushout objects where the images of SE_A,E_A,SE_R,E_R are represented by less than four nodes, we will come up with a situation where there would be either no morphism or no unique morphism between pushout object and arbitrary pushout candidate. An example of this scenario is depicted in the following Figure.
For the example pictured in the above Figure, there is no morphism between *PO* and *Res-3* that makes all the squares in Figure 4.5.3 commute. So, even in this scenario (Less Than Four Nodes), it is proved that the *AllRelax* case is not a pushout construct. So, the proof of above two scenarios combintly proves that, the *AllRelax* case has no pushout. Hence, it is proved that *AllRelax* case has no pushout in category *SysArchsZ*.

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 A.0.1.

For a defined set of production rules (e.g., security introduction, reliability in-

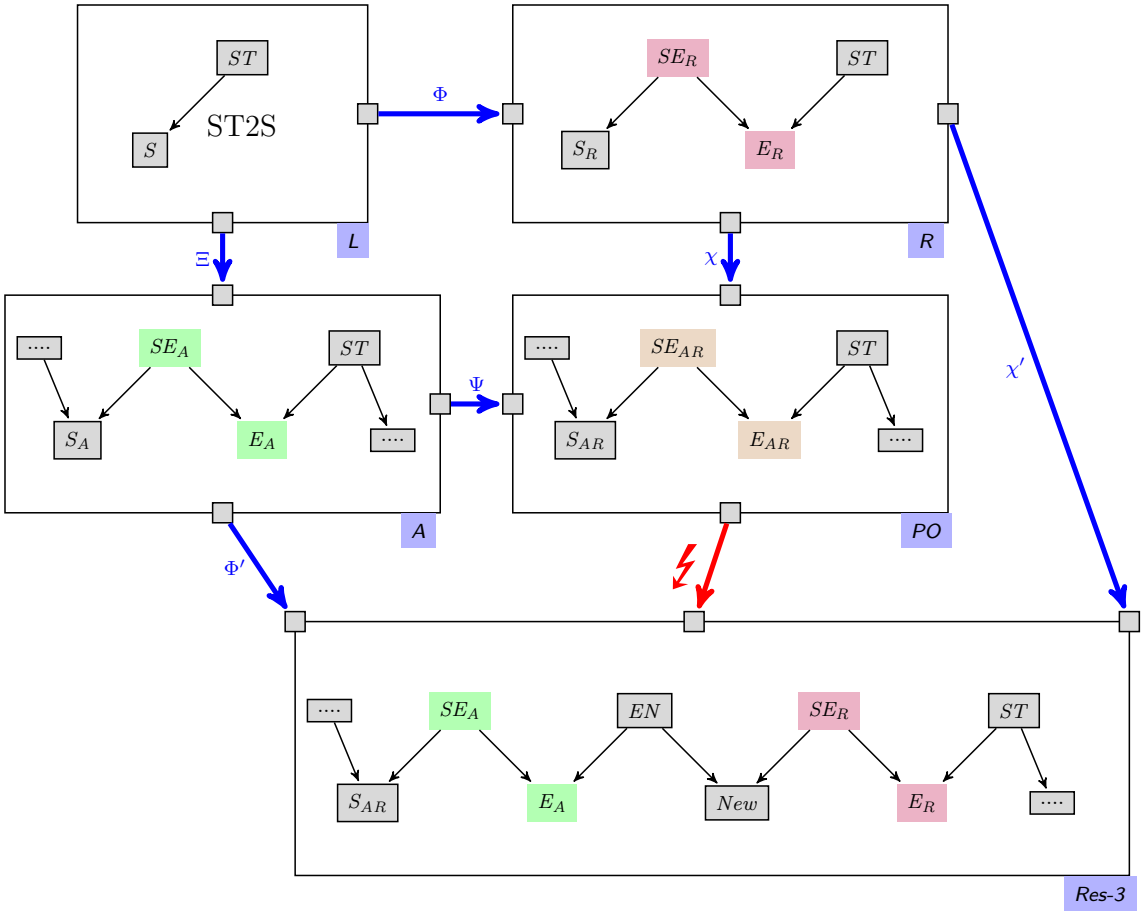


Figure 4.5.3: AllRelax case is SysArchsZ has no pushout

troduction) and given an application architecture, if we apply our transformation technique, in all cases except one (section 4.5), 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.

5 Property Preservation

One of the great advantages of our research work is that it makes the analysis of the system architecture properties feasible. In this chapter, we describe all the possibilities to relate the properties of an old system architecture to the new one. Before going into detail, we review the following essentials.

A system architecture is a diagram in the category *MonoDESC*. A system architecture zigzag homomorphism is a zigzag graph homomorphism of underlying graph that maintains a specification homomorphism between specifications associated with the target vertices, preserves the specifications associated with the source vertices. The properties of a system architecture are the properties of the colimit of the diagram. For more details visit chapter 3.

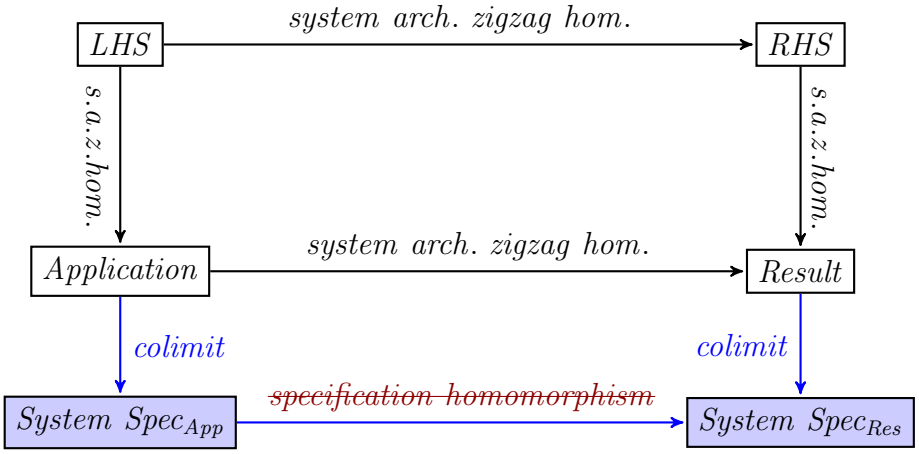


Figure 5.0.1: System Architecture and System Specification

Conformance of the "Result" system architecture with the "Application" system architecture is not straight forward. The above Figure 5.0.1 illustrates this statement: a system architecture zigzag homomorphism between two system architectures, e.g., $Application \xrightarrow{S.A.Z.Hom.} Result$, not necessarily implies the existence of a specification homomorphism between the colimits of these diagrams. Proving properties of "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 automatically check the conformance of the new system architecture ("Result") to the old system architecture ("Application") without any proof obligation. For example, consider sections 5.1 and 5.2.

5.1 Case 1: Specification Homomorphism Exists

After an aspect introduction by applying our transformation technique, in some cases, we might find the existence of a specification homomorphism between the application and the result system specification. Since specification homomorphism preserves the safety and liveness properties (Fiadeiro and Maibaum, 1992), in this case, *system architecture homomorphism* between "Application" and "Result" system architecture will preserve the safety and liveness properties as well. That means, all the properties of the application system specification will be preserved in the result system specification. So, if we prove any safety (partial correctness) and liveness property for an old system architecture ("Application" system architecture), this property will be valid in the "Result" system architecture as well.

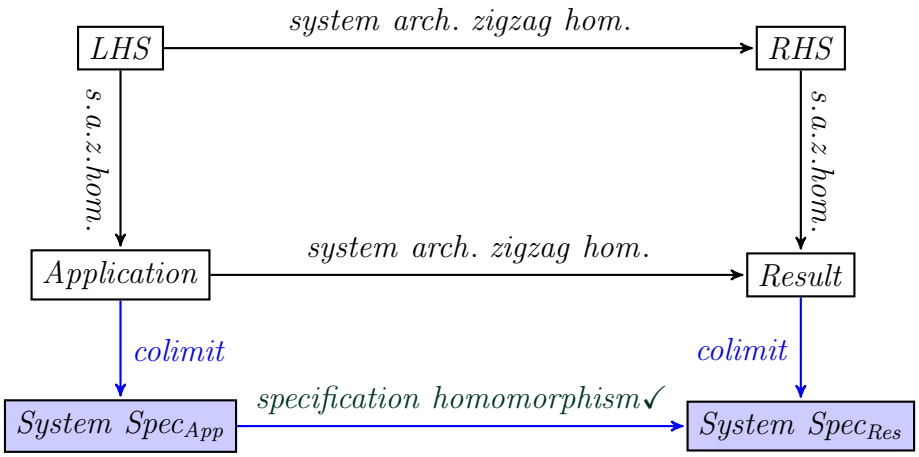


Figure 5.1.1: System Architecture and System Specification

5.1.1 Safety Property

For a given rule (i.e., Security Introduction, Reliability Introduction; explained in appendix B.4) and an "Application" system architecture where an unsecured-unreliable communication exists (find a perfect match of the rules' left hand side in the "Application" architecture), there are a couple of ways we can introduce aspects into it. We can introduce security and reliability aspects in it independently or on top of one another (first security and then reliability or vice-versa) and make the "Result" system architecture secured-unreliable, unsecured-reliable or secured-reliable respectively.

As an illustration of the safety property preservation, consider Figure B.5.3. The top part, i.e., *Left Arch* $\xrightarrow{\Phi}$ *Right Architecture* of this Figure represents a rule, i.e., "Reliability Introduction." Here, the "Left Architecture" represents the system architecture of unsecured-unreliable communication, and the "Right Architecture" represents the system architecture of reliable communication. The "Application Architecture"

in the Figure B.5.3 represents the bigger system architecture of unsecured-unreliable communication. By following our transformation technique, after applying the rule on this given "Application Architecture" we will get the "Result Architecture" that represents the system architecture for reliable communication.

There is a specification homomorphism exists between the "Application System Specification" and the "Result System Specification". The system specifications of the core parts associated with the "Application" and the "Result" system architecture are depicted in appendix B.1.5 and B.3.5.

Depending on the system specification in appendix B.1.5, we can prove a safety property (partial correctness) for the unsecured-reliable architecture (e.g., $\neg\mathbb{F}(msg < 0)$, a message number would never be less than zero). According to the specification homomorphism, this safety property will be valid for the result system architecture, i.e., for the system specification specified in appendix B.3.5 as well.

5.1.2 Liveness Property

Liveness properties are also preserved by specification homomorphism and therefore also by associated system architecture zigzag homomorphism. If there is a specification homomorphism exists between two system specifications, then any specification transformation does not generate proof obligations for the liveness properties desired of the "Result" (Fiadeiro and Maibaum, 1992).

Depending on the system specification specified in appendix B.1.5, we can prove the liveness property (e.g., $Send \rightarrow \mathbb{F}\ Receive$, if a message is sent then eventually it will be received) for the unsecured-unreliable system architecture. Now, if we introduce reliability aspect on top of it and make the architecture unsecured-reliable, this liveness property will automatically hold for the "result architecture".

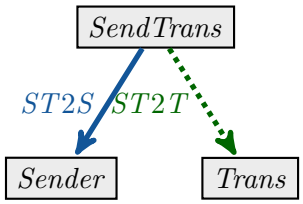
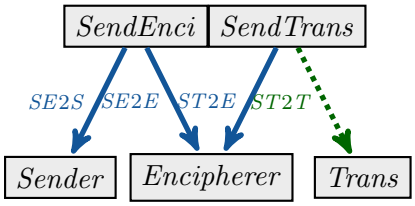
Proposition 5.1.1. *For two given system architectures $\mathscr{A}$ and $\mathscr{B}$, if the zigzag homomorphism $\Psi : \mathscr{A} \rightarrow \mathscr{B}$ is a system architecture homomorphism, then there is a specification homomorphism between the associated system specifications, i.e., all the properties of system architecture $\mathscr{A}$ will be theorems in system architecture $\mathscr{B}$.*

5.2 Case 2: Specification Homomorphism Does Not Exist

A system architecture zigzag homomorphism, e.g., $Application \xrightarrow{S.A.Z.Hom.} Result$, not necessarily implies the existence of a *specification homomorphism* between the associated *colimit* (system specification) of the system architectures. This concept is pictured in the above Figure 5.0.1. For more details consider Figure B.5.2. Let us consider the sub-architecture $Sender \leftarrow SendTrans \rightarrow Trans$ of the "Application Architecture" and the associated sub-architecture $Sender \leftarrow SendEnci \rightarrow Encipherer \leftarrow SendTrans \rightarrow Trans$ in the "Result Architecture". The sub-system specification signatures , i.e., *colimit*, associated to the above sub-architectures are depicted in the

following table. For details definition of morphisms and the components, connectors specifications see appendix B.1 and B.2.

Table 1: An Example of a General Conformance Check

Application sub-architecture	Result sub-architecture
	
Signature Specification	**Signature Specification**
Sort: *Bool, Int* ***Operators:*** *true, false : Bool* *+ : Int, Int → Int* ***Attribute Symbol:*** *rts? : Bool* *msg : Int* *sync_simsg : Int* *rtt? : Bool* *acc_msg : Int* *tra_msg : Int* ***Action Symbol:*** *SyncSendAccept, Prod, Transmit*	***Sort:*** *Bool, Int* ***Operators:*** *true, false : Bool* *+ : Int, Int → Int* *encipher : Int → Int* ***Attribute Symbol:*** *S.rts? : Bool* *S.msg : Int* *sync_scmsg : Int* *E.rts? : Bool* *E.msg : Int* *sync_simsg : Int* *rtt? : Bool* *acc_msg : Int* *tra_msg : Int* ***Action Symbol:*** *SyncSendConcede, Prod, SyncSendAccept, Transmit*

According to the definition of "System Architecture Zigzag Homomorphism" the components *Sender* and *Trans* in "Application Architecture" has a mapping to the components *Sender* and *Trans* in the "Result Architecture". So, the attribute symbol *sync_simsg* in the "application system signature specification" can nondeterministically map to *sync_mcmsg* and *sync_simsg* in "result system signature specification". Similarly, the action symbol *SyncSendAccept* in "application system signature specification" has two potential images *SyncSendConced* and *SyncSendAccept* in the "result system signature specification". Hence, there is no specification homomorphism between the "Application" and the "Result" system specifications. Therefore, the

properties of the old system architecture, i.e., the "Application" system architecture, is not automatically preserved in the "Result" system architecture and hence, generates proof obligations. If we apply the following two techniques, explained in Section 5.2.1,5.2.2, instead of heuristic, proving preservation of safety and liveness properties in the result system architecture will be systematic.

5.2.1 Host Architecture

We can make the proof of property preservations automatic by constructing a host system architecture from the application and the "result system architecture." The idea is depicted in the following Figure 5.2.1.

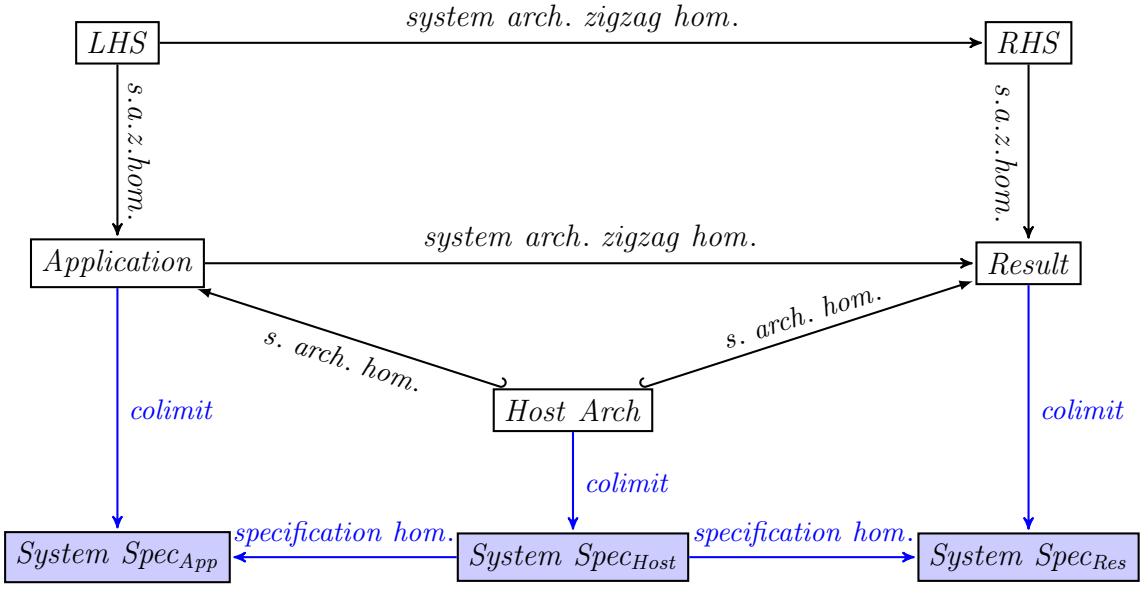


Figure 5.2.1: Host System Architecture and System Specification Homomorphism

The system architecture "Host Arch" is constructed by including all the components in the "Application" system architecture, but only those edges from "Application" system architecture that would remain unchanged or get "extended" in the "Result" system architecture. If we construct the system specification "System $\text{Spec}_{\text{Host}}$," we will find a *specification homomorphism* between the system specification "System $\text{Spec}_{\text{Host}}$" to the system specification "System Spec_{App}" and "System Spec_{Res}" respectively. Since specification homomorphisms preserve safety and liveness properties (Fiadeiro and Maibaum, 1992), all the properties of the "Host Arch" will be preserved in the "Result" system architecture.

So, the properties of any component in "Application Architecture" will be preserved in the "Result Architecture." Similarly, if there is a "system architecture homomorphism" between the associated sub-architectures of the "Application" and the

"Result Architecture", the properties of any sub-architecture in the "Application Architecture" will be preserved in the Result System Architecture.

5.2.2 Proof by Inspection

A system architecture zigzag homomorphism does not necessarily imply the existence of a specification homomorphism between the associated system specifications. That means, we might encounter two associated sub-system specifications, where there is no specification homomorphism exists between them. This concept is illustrated in the following Figure 5.2.2.

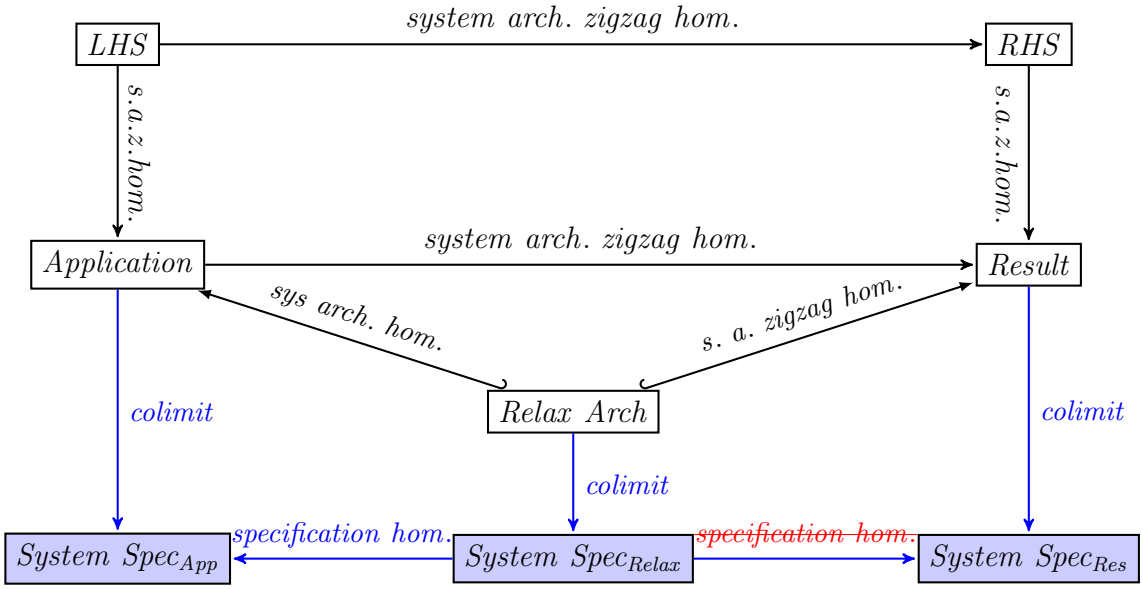


Figure 5.2.2: Glue System Architecture and System Specification Homomorphism

The "Relaxed Architecture" contains only those edges and their associated nodes from "Application Architecture" which get relaxed in the "Result System Architecture." This scenario generates proof obligations.

But rather than starting a rigorous proof technique to prove property preservation (specially safety properties), we might try a "proof by inspection" technique. This technique will allow us to inspect all the attributes in new components which are identified with attributes in old components. If luckily none of the related attributes change their states, then from this inspection we will be able to conclude that the properties will hold. But, if any of the attributes changes its state (most likely), then the property might not be preserved and most likely it will require rigorous proof and this inspection might guide the proof into the right direction. For further illustration consider the sub-architectures and their associated sub-system signature specifications in section 5.2.

The inspection of the unsecured sub-architecture, its components and connectors signature specification gives us the finding where the attribute *send_msg* of component *Sender* identified with the attribute *in_msg* in component *Trans* and they are renamed as *sync_simsg* in the unsecured sub-system specification. On the other hand, in the secured sub-architecture, the attribute *send_msg* of component *Sender* is identified with the attribute *con_msg* in component *Encipherer*, and the attribute *send_msg* of *Encipherer* is identified with the attribute *in_msg* in component *Trans*. The identifications are renamed in secured sub-system specification as *sync_scmsg* and *sync_simsg* respectively. The axioms associated to the sub-system specifications of these sub-architectures are as follows:

unsecured:

S- 1: $\textbf{\textit{Beg}} \rightarrow rts? = false$

S- 2: $\textbf{\textit{Beg}} \rightarrow msg = 0$

S- 3: $(rts? = false \wedge Prod) \rightarrow (\mathbb{X}\, rts? = true \wedge \mathbb{X}\, msg = msg + 1 \wedge \mathbb{X}\, sync\_simsg{=}sync\_simsg)$

S- 4: $(rts? = true \wedge SyncSendAccept) \rightarrow (\mathbb{X}\, sync\_simsg{=}\, msg \wedge \mathbb{X}\, rts? = false \wedge \mathbb{X}\, msg{=}msg)$

S- 5: $(rts? = false) \rightarrow \mathbb{F}\, Prod$

S- 6: $(rts? = true) \rightarrow \mathbb{F}\, SyncSendAccept$

T- 1: $\textbf{\textit{Beg}} \rightarrow rtt? = false$

T- 2: $\textbf{\textit{Beg}} \rightarrow tra\_msg = 0$

T- 3: $(rtt? = false \wedge SyncSendAccept) \rightarrow (\mathbb{X}\, rtt? = true \wedge \mathbb{X}\, acc\_msg = sync\_simsg \wedge \mathbb{X}\, tra\_msg{=}tra\_msg)$

T- 4: $(rtt? = true \wedge Transmit) \rightarrow (\mathbb{X}\, tra\_msg{=}acc\_msg \wedge \mathbb{X}\, rtt? = false \wedge \mathbb{X}\, sync\_simsg{=}sync\_simsg)$

T- 5: $(rtt? = false) \rightarrow \mathbb{F}\, SyncSendAccept$

T- 6: $(rtt? = true) \rightarrow \mathbb{F}\, Transmit$

Secured:

S- 1: $\boldsymbol{Beg} \rightarrow rts? = false$

S- 2: $\boldsymbol{Beg} \rightarrow msg = 0$

S- 3: $(rts? = false \wedge Prod) \rightarrow (\mathbb{X}\, rts? = true \wedge \mathbb{X}\, msg = msg + 1 \wedge \mathbb{X}\, sync\_scmsg{=}sync\_scmsg)$

S- 4: $(rts? = true \wedge SyncSendConcede) \rightarrow (\mathbb{X}\, sync\_scmsg{=}\, msg \wedge \mathbb{X}\, rts? = false \wedge \mathbb{X}\, msg{=}msg)$

S- 5: $(rts? = false) \rightarrow \mathbb{F}\, Prod$

S- 6: $(rts? = true) \rightarrow \mathbb{F}\, SyncSendConcede\}$

E- 1: $\boldsymbol{Beg} \rightarrow E.rts? = false$

E- 2: $\boldsymbol{Beg} \rightarrow E.msg = 0$

E- 3: $(rts? = false \wedge SyncSendConcede) \rightarrow (\mathbb{X}\, E.rts? = true \wedge \mathbb{X}\, E.msg = encipher(sync\_scmsg) \wedge \mathbb{X}\, sync\_simsg{=}sync\_simsg)$

E- 4: $(E.rts? = true \wedge SyncSendAccept) \rightarrow (\mathbb{X}\, sync\_simsg{=}\, E.msg \wedge \mathbb{X}\, E.rts? = false \wedge \mathbb{X}\, E.msg{=}E.msg \wedge \mathbb{X}\, sync\_scmsg{=}sync\_scmsg)$

E- 5: $(E.rts? = false) \rightarrow \mathbb{F}\, SyncSendConcede$

E- 6: $(E.rts? = true) \rightarrow \mathbb{F}\, SyncSendAccept$

T- 1: $\boldsymbol{Beg} \rightarrow rtt? = false$

T- 2: $\boldsymbol{Beg} \rightarrow tra\_msg = 0$

T- 3: $(rtt? = false \wedge SyncSendAccept) \rightarrow (\mathbb{X}\, rtt? = true \wedge \mathbb{X}\, acc\_msg = sync\_simsg \wedge \mathbb{X}\, tra\_msg{=}tra\_msg)$

T- 4: $(rtt? = true \wedge Transmit) \rightarrow (\mathbb{X}\, tra\_msg{=}acc\_msg \wedge \mathbb{X}\, rtt? = false \wedge \mathbb{X}\, sync\_simsg{=}sync\_simsg)$

T- 5: $(rtt? = false) \rightarrow \mathbb{F}\, SyncSendAccept$

T- 6: $(rtt? = true) \rightarrow \mathbb{F}\, Transmit$

If we analyze the axioms of the secured sub-system specification, we see that the attribute *sunc_scmsg* changes its state in axiom $E-3$. So, the safety property for unsecured sub-system architecture, e.g., $\neg\mathbb{F}(msg < 0)$, might not hold in the secured sub-system architecture.

5.3 Feature Interaction Detection

Another advantage of our work is that we will be able to detect feature interaction if any occurs, due to adding several features on top of one another.

As an example consider Figure 5.3.1, where a security aspect is introduced on top of a reliable communication. For further illustration consider the specifications of the components *TransPlus* and *Monitor* in appendix B.3, and *Encipherer* in appendix B.2.

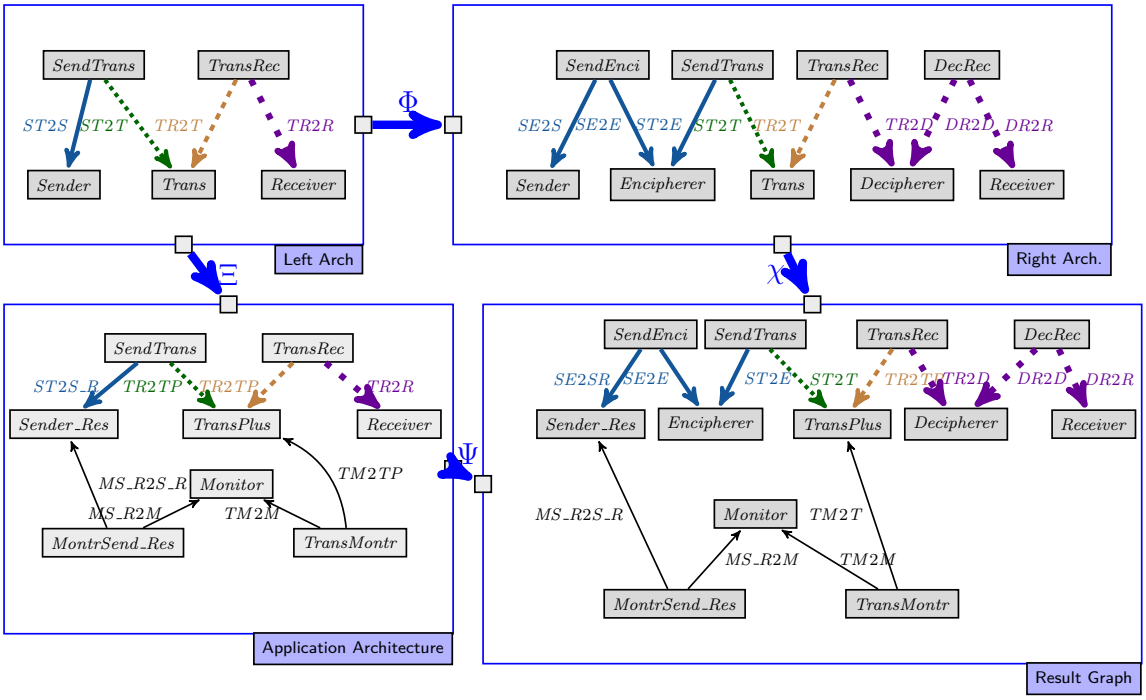


Figure 5.3.1: Reliable-Secured Communication

The *TransPlus* checks for errors in received messages and sends error-free messages to an appropriate receiver (i.e., *Decipherer*), but sends both error-prone and error-free messages to the *Monitor*. The *Monitor* has the capacity to check whether the message it receives is the expected one or not. In case of expected messages, it updates its accepted message status; otherwise, it sends a request to the *Sendr_Res* component to re-send the lost message.

Depending on the above component specifications, we can prove (e.g., *Send* $\rightarrow \mathbb{F}$ *Receive* if a message is sent then eventually it will be received) liveness property, for the unsecured-reliable, i.e., application, system architecture. However, if we introduce security on top of it and make the architecture secured-reliable, this liveness property might not hold for the result architecture. For further illustration consider the specification of the component *Encipherer*.

An *Encipherer* enciphers a received message and sends it to an appropriate receiver (i.e., *Trans*). Initially, the *msg* is 0 and *rts?* status is false. After *Encipherer* concedes the message from an appropriate source and enciphers it, the *rts?* becomes true and the value of *msg* is changed. Finally, after the execution of the *Send* action the *rts?* becomes false again. The properties of the *Encipherer* can be represented by the following axioms:

If we analyze the "Result Architecture" in the Figure 5.3.1, we can see that the *Monitor* will receive enciphered messages and these are always different from the expected messages. Therefore, *Monitor* will keep requesting the *Sender_Res* to re-send the lost message and there will be a deadlock.

In summary, aspect introductions at the *software architecture* level by performing *zigzag graph transformations* enable us to prove the desired safety and liveness properties. Besides that, we will be able to detect deadlocks if there are any. The most important essence of our proof technique is modularity, i.e., we can use modularity to our advantage in investigating (preserving) safety and liveness properties.

6 Tool

In order to turn our theory into practice, we have developed tool support. The tool is implemented based on Hets, an initiative of Mossakowski et al. (2007). Though our theory is independent of any logic, we have implemented our MFLogic language, and it is implemented as an extension of Casl, a contribution of Astesiano et al. (2002). This tool will allow us to parse and check the syntactical correctness of our specifications, i.e., the MFLogic specifications. Moreover, this tool also checks the correctness of the specification in terms of the *static semantics*. Our primary goal is to provide support for performing and analysing aspect introduction at the system architecture level, where system architectures are diagrams in the category of *DESC* (Proposition 2.5.11), and aspect introduction is performed via the *zigzag graph transformation* sketched in Chapter 4. Our tool supports our primary goal. In summary, for a given aspect introduction rule and an application architecture, this extended Hets can semi-automatically generate the "result architecture" by performing a zigzag transformation. For any given system architecture, this tool has the capacity to generate the *colimit* and, hence, the *system architecture specification*. Analysing the properties of the system specification, i.e., the conformance check of the new architecture with the old architecture and proving a newly introduced property in the result architecture is a work in progress / potential future work. Hets and Casl have been briefly introduced in section 6.1 and 6.2. In the following section we talk about the implementation details of MFLogic, followed by some essentials of the tool. Finally the last section contains a guided tour, namely "how to use it?".

6.1 Hets

Hets (Heterogeneous Tool Set) is an integration tool that is formal, flexible, and multi-lateral. It provides parsing, static analysis, and proof management facilities for heterogeneous specifications by incorporating various provers and different specification languages. In Hets, logic translation is being treated as a first-class citizen. For more details visit Mossakowski et al. (2007).

6.2 CASL

The Common Algebraic Specification Language (Casl) is the heart of a *family* of languages. In other words, it is a consolidation of past work on the design of algebraic specification languages for the formal specification of functional requirements and modular design of software. It has been designed by CoFI, the international Common Framework Initiative for algebraic specification and development. Casl consists of several major parts, which are: basic specification, structured specification, architectural specification, and specification libraries.

A Casl basic specification consists of declaration and axioms. A declaration

introduces signature components and the axioms describe the properties of the models of the specification.

A CASL architectural Specification is used to describe the modular structure of software and is probably considered as the most novel aspect of CASL. A CASL architectural specification consists of *unit declarations* and a *unit expression*. The *unit expression* explains how the units are to be combined and a *unit expression* represents a component with its specification. For further references please read Astesiano et al. (2002).

6.3 Implementation Details

6.3.1 MFLogic

To get an overall idea about MFLogic, the reader should review subsection 2.5 and for more details follow Fiadeiro and Maibaum (1992).

In order to implement MFLogic in HETS, we have implemented MFLogic as an extension of CASL, which allows us to use the designed-for-extension CASL infrastructure of HETS, and also re-use much of the operator and predicate symbol infrastructure for attribute and action symbols.

A temporal theory following Fiadeiro and Maibaum (1992) is formulated over a signature $\theta = (\Sigma, A, \Gamma)$, where Σ is a conventional algebraic signature, A is a separate finite set of *attribute symbols* that syntactically behave like the function/operator symbols of Σ, and Γ is a separate set of *action symbols* that syntactically behave like predicate symbols. While the interpretation of Σ is a conventional algebra, the semantics of attribute and action symbols is parameterised over time (encoded as natural numbers).

The temporal part of the logic has been implemented as an extension of CASL formulae by adding term and formula constructors. The term language is additionally enriched with a "next" operator, such that $\mathbb{X}\ t$ refers to "the value of t in the next instant." The formula language is enriched with conventional LTL operators and the new atomic formula **BEG** that holds only at time point 0.

We use the "views" of CASL/HETS to encode morphisms and can generate the proof obligations, including locality preservation, for checking the well-definedness of the resulting morphisms.

6.3.2 Description of the Modules

Implementation of MFLogic in HETS consists of several modules. An overview of each module, important code snippets from them, and the associated description of the code is explained in the following section.

AS_MFLogic:

This module contains the abstract syntax for MFLogic. MFLogic is an extension of CASL. Only the added (extended) syntax data types are defined in this module. The definition of the extended signature items and extended formulae are given as follows:

```
-- | Extended signature item only

data MF_SIG_ITEM  = Att_items [Annoted (OP_ITEM MF_FORMULA)] Range
 ↪  -- Attribute Signature
                  | Actn_items [Annoted (PRED_ITEM MF_FORMULA)] Range
 ↪  -- Action Signature
                    deriving (Show, Typeable, Data)

-- | Extended formulas

data MF_FORMULA
    -- Formula constructors:
    = FormX (FORMULA MF_FORMULA) Range    -- X, i.e., next applied to
 ↪  a formula is a formula
    | FormF (FORMULA MF_FORMULA) Range    -- F, i.e., future applied
 ↪  to a formula is a formula
    | Action PRED_SYMB [TERM MF_FORMULA] Range  -- Action predicate
 ↪  is a formula
    | BEG Range -- Begining of time is a formula
    -- Term constructors:
    | AttrSel OP_SYMB [TERM MF_FORMULA] Range    -- Attribute
 ↪  application
    | TermX (TERM MF_FORMULA) Range             -- X applied to Term.
    deriving (Show, Eq, Ord, Typeable, Data)
```

ATC_MFLogic:

Generates an instance of ShATermConvertible. "The tool *genRules* provides the ability to create rules automatically, which can be processed by *DrIFT*. The directory *ATC* includes the files with ATermConvertible instances, which are necessary to read datatypes from and to write them to files in the Shared-ATerm-format." For further details read `https://github.com/spechub/Hets/blob/master/ATC/doc/ATC-Rule-Generation.tex` or `/local/hets/ATC/doc/ATC-Rule-Generation.tex`. Here "local" is the name of the directory that contains HETS installation.

ComputeZigZagTrans:

For a given diagram (*DevGraph*), this module collects the description (specification) of the components, gets the morphisms, the system architecture (Set of nodes and morphisms of a diagram), the system architecture zigzag homomorphism, extracts the span, calculates the cospan by performing the zigzag graph transformation, implements different helper functions to perform the transformation and finally returns the result architecture (*SysArchReal*), by saving it into a file. Some of the important type definitions are as follows:

```
-- | nodeName or SpecName type

type MF_SPEC_NAME = String -- ^ IRI too complicated for now

type MF_DIAGNODE_NAME = String  -- ^ Atchitecture name Type, i.e., L,
 ↪  R, A

type MF_MOR_NAME = String -- ^ Morphism name type

-- | System Architecture Names

data SysArch = SysArch
  { sa_nodes :: Set MF_SPEC_NAME  -- ^ A set of Node Names
  , sa_edges :: Set MF_MOR_NAME   -- ^ A set of Morphism Names

  }deriving (Show)

-- | System Architecture with actual specification and morphism

data SysArchReal obj mor = SysArchReal
  { sa_specs :: obj   -- ^ A map from Node Name to it's
 ↪  specification,
                  -- e.g., Map MF_SPEC_NAME (MFSign, ([Annoted
 ↪  (FORMULA MF_FORMULA)]))
  , sa_mors :: mor    -- ^ A map from morphism Name to it's morphism,
                      --  e.g.,(Map MF_MOR_NAME MorphismMF)

  }

{- | System Architecture Zpath Homomorphism consists of node Name map
 ↪  and edge Name map.
The sazh0_nodeMap  maps Node Name of an architecture to the pair of a
 ↪  'target' node Name
```

```
and  'morphism' Name of  another architecture. The sazh0_edgeMap is a
 ↪  map between two
morphism Names. Here the type Bool represent the direction.
 ↪  sazh0_edgeMap can be of
two types:
a) single morpshim Name to single morphism Name map (Signleton
 ↪  list),
e.g., "ST2S_L"  [("ST2X_R", True)] or
b) a single morphism to zigzag path map,
e.g., "ST2S_L"  [("ST2X_R", True), ("Q2X_R", False), ("Q2S_S",
 ↪  True)]
-}

data SysArchZPHom0 = SysArchZPHom0
  { sazh0_nodeMap :: Map MF_SPEC_NAME (MF_SPEC_NAME, MF_MOR_NAME)
  , sazh0_edgeMap :: Map MF_MOR_NAME [(MF_MOR_NAME, Bool)]
  }deriving (Show)

{-|
System Architecture Zpath Homomorphism consists of spec map and
 ↪  morphism map.
The |sazh_nodeMap|  maps node Name of an architecture to the node
 ↪  Name ,
node sprcification, morphism Name and the morphism of  another
 ↪  architecture.
The |sazh_edgeMap| is a map between a morphism Name to a list of
 ↪  morphism Name,
the morphsim and the direction in another architecture. Oviously, it
 ↪  can be of
two types:
a) single to single, e.g., "ST2S_R"  [("ST2S_R_B", morphism, True)]
 ↪  or
b) single morphism to zigzag path, e.g.,
"ST2S_R"  [("ST2S_R_B", morphism, True), ("Q2X_R_B", morphism,
 ↪  False),
("Q2S_R_B", morphism, True)]
-}

data SysArchZPHom obj= SysArchZPHom
  { sazh_nodeMap :: Map MF_SPEC_NAME ((MF_SPEC_NAME, obj),
 ↪  (MF_MOR_NAME, MorphismMF))
 , sazh_edgeMap :: Map MF_MOR_NAME [(MF_MOR_NAME, MorphismMF,Bool)]
```

```
  }deriving (Show)

-- | the 'SysArchNodeLabel' is a record consisting three record
 ↪  elements, i.e.,
-- | the basic specification, the range and
-- | the extended signature (plain signature + symbol)

data SysArchNodeLabel = SysArchNodeLabel
  { sanl_BasicSpec :: MF_BASIC_SPEC
  , sanl_BasicSpecRange :: Range
  , sanl_ExtSig :: ExtSignMF
  } deriving (Show)

{- |
The following |MF_SYSARCH_DIAGRAM|  data type consists of the
folowing four record elements:
a) |mfsaDiag_allSpecs| : map all spec name to its specificaiton
for all the architectures, i.e., L, R, A, .....

b) |mfsaDiag_allMors|: all the morphism maps between two
  architecture, L -> A, L -> R, P0-> A,
  P1-> A ....

c) |mfsaDiag_nodes|: The set of node names and morphism names of all
 ↪  architectures, e.g.,
L -> Record (Set, Set), R -> Record (Set, Set) , ....

d) |mfsaDiag_edges|: ZedHomomorphism between two
architecture, e.g.,
(L,A) -> Zhom, (L,R) -> Zhom, .....

-}

data MF_SYSARCH_DIAGRAM = MF_SYSARCH_DIAGRAM
  { mfsaDiag_allSpecs :: Map  MF_SPEC_NAME SysArchNodeLabel
  , mfsaDiag_allMors :: Map MF_MOR_NAME ((MF_SPEC_NAME,
 ↪  MF_SPEC_NAME), MorphismMF)
  , mfsaDiag_nodes :: Map MF_DIAGNODE_NAME SysArch
  , mfsaDiag_edges ::  Map (MF_DIAGNODE_NAME, MF_DIAGNODE_NAME)
 ↪  SysArchZPHom0
  }
```

Locality:

Locality axioms in Fiadeiro and Maibaum (1992) are implicit axioms. The preservation of locality axioms needs to be checked for morphisms.

Logic_MFLogic:

In this module, we made the logicMFLogic an instance of the *Language*, *SignExtension*, *Syntax*,*Sentences*, *StaticAnalysis*, *ProjectSublogicM*, *Logic* classes and have implemented different helper functions associated with those classes.

```
-- | the name of a logic.
-- | Making MFLogic an instance of Language class defined in
 ↪  Logic/Logic.hs
instance Language MFLogic where
  ........

-- | Making SignExtMF an instance of SignExtension class defined in
 ↪  CASL/Sign.hs
-- | Extended signature

instance SignExtension SignExtMF where
  ........

-- | Abstract syntax, parsing and printing.
-- | Making an instance of Syntax class defined in Logic/Logic.hs
instance Syntax MFLogic MF_BASIC_SPEC
                MFSymbol SYMB_ITEMS SYMB_MAP_ITEMS
      where
       ...........

-- | Making an instance of Sentences class defined in Logic/Logic.hs

instance Sentences MFLogic MFLogicFORMULA
           MFSign MorphismMF MFSymbol
     where
     ............

-- | static analysis of basic specifications and symbol maps.
-- | Making an instance of StaticAnalysis defines in Logic/Logic.hs

instance StaticAnalysis MFLogic MF_BASIC_SPEC MFLogicFORMULA
        SYMB_ITEMS SYMB_MAP_ITEMS
```

```
        MFSign
        MorphismMF
        MFSymbol Mor.RawSymbol where
      ....................................

-- | class providing a partial projection of an item to a sublogic
-- | defined in Logic/Logic.hs

instance ProjectSublogicM MFLogic_Sublogics MFSymbol
       where
        ......................

-- | Making an instance of Logic class defined in
 ↪  Logic/Logic.hsLogic
-- | The central type class of Hets.

instance Logic MFLogic MFLogic_Sublogics
               MF_BASIC_SPEC MFLogicFORMULA SYMB_ITEMS SYMB_MAP_ITEMS
               MFSign
               MorphismMF
               MFSymbol Mor.RawSymbol () where
         ..............................
```

MFLogicSign:

The signatures for MFLogic are an extension of CASL signatures. The signatures also serve as local environments for the basic static analysis. This module defines several data types and helper functions associated with the MFLogic signature. The two important data types are *MFMap* and *SignExtMF* and are defined as follows:

```
{- |
   MFMap| is going to be used as the |m| parameter.
   We follow the convention used (apparently) in |CASL.Morphism|
 ↪  that
   symbols outside the domain of these |Map|s are considered as
 ↪  mapped
 to themselves.
-}

data MFMap = MFMap
  {
    attr_MapMor :: Mor.Op_map
  , actn_MapMor :: Mor.Pred_map
```

```
  } deriving (Eq, Ord, Show)

{-|
MFsign is defined in StatAna.hs
SignExtMF is the extended part of the signature only
-}

data SignExtMF = SignExtMF
  { attrSig :: OpMap   -- attrSig
  , actnSig :: PredMap
  } deriving (Show, Eq, Ord, Typeable, Data)
```

MFSymbol:

This module defines the MFLogic symbols as an extension of CASL logic symbols. This module also contains several helper functions, and some of them are: *symbolForgetMF*, *splitMFSymbols*, *caslSymbolToMF*. The name of the functions reflects their meaning.

```
data MFSymbType
  = CASLSymbType CASLSign.SymbType
   | AttrAsItemType CASLSign.OpType
  {- since symbols do not speak about totality, the totality
     information in OpType has to be ignored -}
   | ActnAsItemType CASLSign.PredType
   deriving (Show, Eq, Ord, Typeable, Data)


data MFSymbol  = GenSymbol
   { symName :: Id
   , symbType :: MFSymbType
   }
   deriving (Show, Eq, Ord, Typeable, Data)
```

Parse_AS:

This module implementation works as a *Parser* for the MFLogic.

Print_AS:

This module works as a printer for the MFLogic by printing the data types in *AS_MFLogic* and *MFLogicSing*.

StatAna:

This module name reflects its meaning, i.e., static analysis of the MFLogic.

Sublogic:

This module provides the sublogic functions (as required by Logic.hs) for MFLogic. It is based on the respective functions for CASL.

6.4 Installation and Some Other Essentials

The development of our tool is based on HETS. So, the very first step to use our tool is to get/install HETS.

6.4.1 Installing Hets

One of the ways to install the HETS is to build it from source. In order to get a copy of HETS, the convenient way is to follow the instructions in the "Build from source" section of the link `https://github.com/spechub/Hets/blob/master/README`. The next step is to run all available tests against the HETS binary by writing *make check* in your terminal (any shell). The last step is to run HETS. One way to run HETS is via a web interface. In order to run HETS as a web server one needs to use either of the options *-X* or *–version*. There are two options available to list the specification libraries, and they are *-L DIR* and *–hets-libdir=DIR*. A sample command to run HETS via the web interface and to list specification libraries (hets-lib) from the home directory in which HETS has been installed is: *.../hets$ ./hets -X -L /usr/local/packages/Hets/lib/hets/hets-lib*. After entering the above command in any shell terminal, the message "hets server is listening on port 8000" ensures that HETS is ready to run. Open your favorite web browser and write the url `http://localhost:8000` to display all the specification libraries. For further details read the HETS user guide by Mossakowski et al. (2013), specially Section 8 to Section 10.

6.4.2 Installing MFLogic

Implementation of the language MFLogic is available on-line. Like other languages, i.e., *CoCASL*, *CASL*, *HasCASL*, *DOL* create an MFLogic directory inside the *hets* directory and clone the content of the *GitHub* repository from the link: `https://github.com/mnhminto/MFLogic.git`.

To connect MFLogic into Hets, we need to create some fresh files along with modifying or extending a couple of others. All those files are kept inside the *otherFiles* directory of the *GitHub* repo. It is required to place those files into the appropriate file locations. The location of each file, along with the complete code or the code snippet for them are listed below.

..../hets/Common/Keywords.hs

The keywords require for MFLogic are:

```
attS :: String
attS = "attr"

actnS :: String
actnS = "actn"

begS :: String
begS = "BEG"

nextsS :: String
nextsS = "X"

futureS :: String
futureS = "F"
```

..../hets/Comorphisms/CASL2MFLogic.hs

The *CASL2MFLogic* module is a newly created module. The content of the module is:

```
{-# LANGUAGE MultiParamTypeClasses, TypeSynonymInstances,
 ↪  FlexibleInstances #-}
{- |
Module      :  ./Comorphisms/CASL2MFLogic.hs
Description :  embedding from CASL to MFLogic
Copyright   :  (c) Md Nour Hossain, Wolfram Kahl, 2017
License     :  GPLv2 or higher, see LICENSE.txt

Maintainer  :  hossaimn@mcmaster.ca
Stability   :  provisional
Portability :  non-portable (imports Logic.Logic)

The embedding comorphism from CASL to MFLogic.

-}

module Comorphisms.CASL2MFLogic where

import Logic.Logic
import Logic.Comorphism
```

```
import qualified Data.Set as Set
import Common.ProofTree

-- CASL
import CASL.Logic_CASL
import CASL.Sublogic as SL
import CASL.Sign
import CASL.AS_Basic_CASL
import CASL.Morphism

-- MFLogic
import MFLogic.Logic_MFLogic
import MFLogic.AS_MFLogic
import MFLogic.MFLogicSign
import MFLogic.StatAna (MFSign)
import MFLogic.Sublogic
import MFLogic.MFSymbol(MFSymbol, caslSymbolToMF)

-- | The identity of the comorphism
data CASL2MFLogic = CASL2MFLogic deriving (Show)

instance Language CASL2MFLogic -- default definition is okay

instance Comorphism CASL2MFLogic
               CASL CASL_Sublogics
               CASLBasicSpec CASLFORMULA SYMB_ITEMS SYMB_MAP_ITEMS
               CASLSign
               CASLMor
               Symbol RawSymbol ProofTree
               MFLogic MFLogic_Sublogics
               MF_BASIC_SPEC MFLogicFORMULA SYMB_ITEMS SYMB_MAP_ITEMS
               MFSign
               MorphismMF
               MFSymbol RawSymbol () where
    sourceLogic CASL2MFLogic = CASL
    sourceSublogic CASL2MFLogic = SL.top
    targetLogic CASL2MFLogic = MFLogic
    mapSublogic CASL2MFLogic s = Just \$ s { ext_features = False }

    map_theory CASL2MFLogic = return . embedCASLTheory
 ↪  emptyMFLogicSign
```

```
    map_morphism CASL2MFLogic = return . mapCASLMor emptyMFLogicSign
 ↪  emptyMFMap
    map_sentence CASL2MFLogic _ = return . mapFORMULA
    map_symbol CASL2MFLogic _ = Set.singleton . caslSymbolToMF
    has_model_expansion CASL2MFLogic = True
    is_weakly_amalgamable CASL2MFLogic = True
    isInclusionComorphism CASL2MFLogic = True
```

..../hets/Comorphisms/LogicGraph.hs

The *comorphismList* function has been updated by adding *CASL2MFLogic*.

```
comorphismList :: [AnyComorphism]
comorphismList =
    [ .......
#ifdef CASLEXTENSIONS
    ,.........
    , Comorphism CASL2MFLogic
    ,...........
    ]
```

..../hets/Comorphisms/LogicGraph.hs

In *LogicGraph* module the *logicList* function definition get updated by appending *MFLogin* into the list.

```
logicList :: [AnyLogic]
logicList =
  [ ......
#ifdef CASLEXTENSIONS
  , Logic MFLogic
  ,......
  ]
```

..../hets/Interfaces/CmdAction.hs

The module *MFLogic.ComputeZigZagTrans* gets imported and the function call *zigZagGraphTransformation* gets included in the *globLibResultAct* function definition.

```
import MFLogic.ComputeZigZagTrans (zigZagGraphTransformation)

globLibResultAct :: [(GlobCmd, LibName -> LibEnv -> Result LibEnv)]
globLibResultAct =
  [ ..........
```

```
  , (ZigZagGraphTransformation, zigZagGraphTransformation)
  ]
```

..../hets/Interfaces/Command.hs

In *Command* module the *GlobCmd* data type and *menuTextGlobCmd*, *cmdlGlobCmd* and *describeGlobCmd* functions get modified.

```
data GlobCmd =
  ........
  | ZigZagGraphTransformation
    deriving (Eq, Ord, Enum, Bounded, Show)

-- list of command names in the gui interface
menuTextGlobCmd :: GlobCmd -> String
menuTextGlobCmd cmd = case cmd of
  .........
  ZigZagGraphTransformation -> "ZigZag-Graph-Transformation"

-- | even some short names for the command line interface
cmdlGlobCmd :: GlobCmd -> String
cmdlGlobCmd cmd = case cmd of
  ..........
  ZigZagGraphTransformation -> "zigzag-gt"
  .......

describeGlobCmd :: GlobCmd -> String
describeGlobCmd c =
  ......
  ZigZagGraphTransformation -> "Perform Zigzag Graph Transformation"
  ......
```

..../hets/Makefile

The *Makefile* required the following adjustments to support the MFLogic implementation in HETS.

```
# the list of logics that need ShATermConvertible instances
logics = CASL HasCASL Isabelle Modal Hybrid TopHybrid Temporal \
    CoCASL MFLogic COL CspCASL CASL_DL \

# files generated by DriFT
drifted_files = Common/AS_Annotation.hs \
```

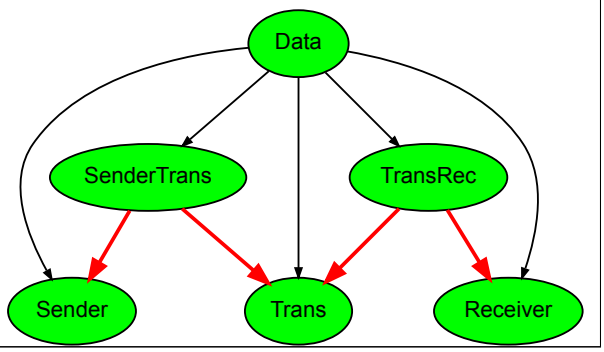
```
    CASL/AS_Basic_CASL.hs Modal/AS_Modal.hs Hybrid/AS_Hybrid.hs
 ↪  TopHybrid/AS_TopHybrid.hs  \
    Syntax/AS_Structured.hs Syntax/AS_Architecture.hs
 ↪  Syntax/AS_Library.hs \
    Propositional/AS_BASIC_Propositional.hs \
    CoCASL/AS_CoCASL.hs MFLogic/AS_MFLogic.hs  COL/AS_COL.hs \

MFLogic_files = MFLogic/AS_MFLogic.hs MFLogic/MFLogicSign.hs
 ↪  MFLogic/MFSymbol.hs

MFLogic/ATC_MFLogic.der.hs: $(MFLogic_files) $(GENRULES)
        $(GENRULECALL) -i CASL.ATC_CASL -o $@ $(MFLogic_files)
```

If all of the above has been completed successfully, from the *hets* directory run the *make* (*.../hets$ make*) command to compile and link HETS.

6.5 How to Use?

The very first step to using HETS is to install HETS on the local machine, and then include the MFLogic language implementation and make sure the extended HETS is running in a browser. If any of the above has not been done already, complete it by following the required steps in Section 6.4. The very next important step is to write the MFLogic specification. Our MFLogic is an extension of CASL. So, to learn MFLogic specification writing one needs to know how to write CASL specifications and a potential source to learn about CASL specification is through the CASL user and reference manual, Bidoit and Mosses (2003), Mosses (2004). The following specification is an example of an MFLogic specification. This sample example also answers the question about how to write the extended part of the MFLogic specification.
The morphism between two component specifications (in the same architecture or between two different architectures) is represented as a CASL view, and an example of a view is:
Since our goal includes system architecture transformation via pushouts of the "zigzag system architecture homomorphisms" introduced in Chapter 4, we need to represent diagrams of system architectures, that is, diagrams of diagrams of object descriptions. We currently achieve this by collecting each system architecture into a HETS architectural specification and extracting the system architecture homomorphisms from the architecture-crossing views. As an example, the simple sender-receiver communication system architecture is represented below:

arch spec *L* =
units *D* : Data;
ST : SenderTrans;
TR : TransRec;
S : Sender;
T : Trans;
R : Receiver
result *S* **and** *R*
end



Identifying a zigzag edge requires creating an auxiliary system architecture where each unit specification is an empty specification. To relate an auxiliary system architecture with a zigzag edge requires creating system architecture homomorphisms between all the edges of the zigzag edge and the edges of the auxiliary architecture. Each edge in the auxiliary architecture has been mapped to an individual edge of the zigzag edge. The map preserves the direction of the edges and all the mapped images in combination represent a zigzag edge.
Now it is time to get the results of your handiwork. To introduce an aspect, represent all the three architectures, i.e., L, R, A, as CASL architectural specifications, specify all the components, specify all the morphisms, represent all of the zigzag edges and save the specification file into MFLogic library.
After running the HETS web server (write the command `./hets-X-L/usr/local/packages/Hets/lib/hets/hets-lib` in any shell terminal in the *hets* directory) and loading it in a web browser (`www.localhost:8000`) will display the list of logic libraries. If we load the specification file by browsing the logic library, e.g., MFLogic and no issues are encountered, we will be able to see the development graph.
If we load a specification file which contains:

- A graph transformation rule, i.e., left and right architecture specification, component specifications, morphisms, system architecture homomorphisms.
- An application architecture, i.e., component specification, architectural specification and morphisms.
- All auxiliary architectures.
- All zigzag edge representations.

clicking on the zigzag-gt from the commands list will automatically generate the resulting architecture by introducing the desired aspect into the application architecture.

```
library arch

logic MFLogic

spec Data =
        sorts Bit, Int
        ops    0, 1 : Int
        ops    on, off : Bit
end

spec SenderTrans =
        Data
then attr sync_simsg : Int
        actn SyncSendAccept : Bit
end

spec Sender =
        Data
then op      __+__ : Int * Int -> Int
        attrs rts : Bit;
                  msg : Int;
                  send_msg : Int
        actns Prod : Bit;
                  Send : Bit
        . BEG => rts = off
        . BEG => msg = 0
        . rts = off /\ Prod(on)
           => X (rts = on /\ X (msg = msg + 1 /\ X (send_msg = send_msg)))
        . rts = on /\ Send(on)
           => X (send_msg = msg /\ X (rts = off /\ X (msg = msg)))
        . rts = off => F (Prod(on))
        . rts = on => F (Send(on))
end

view ST2S :
        SenderTrans to Sender =
        sync_simsg |-> send_msg, SyncSendAccept |-> Send
end
```

Loading the result architecture file through the web server will provide us the development graph, i.e., the resulting architecture.
Clicking on the compute-colimit command will generate the system specification (*colimit*) for the development graph, e.g., the "result system architecture".

```
%Zhom between Right Hand side
%and P0

view SP02R : SenderP0 to SenderR
end

view SEP02R : SendEnciP0 to SendEnciR
end

view CP02R : CiphererP0 to CiphererR
end

view STP02R : SenderTransP0 to SenderTransR
end

arch spec P0 =
        units Em : Empty;
                S : SenderP0;
                SE : SendEnciP0;
                E : CiphererP0;
                ST : SenderTransP0
        result ST and S
end
```

Hets v0.99, 1529450663

Hompage: hets.eu **Contact:** hets-devel@informatik.uni-bremen.de

Choose a display type: default svg xml json dot symbols session pp.het pp.tex pp.xml pp.html pdf

internal command overview as XML: menus

Select a local file as library or enter a HetCASL specification in the text area and press "submit", or browse through our Hets-lib library below.

Choose file No file chosen

submit

/./

Basic/

CASL/

Calculi/

CaseStudies/

CoCASL/

CommonLogic/

Conservativity/

ConstraintCASL/

library basic svg xml json dot symbols session pp.het pp.tex pp.xml pp.html pdf menus
xupdate Choose file No file chosen *impacts* Choose file No file chosen submit

tools:

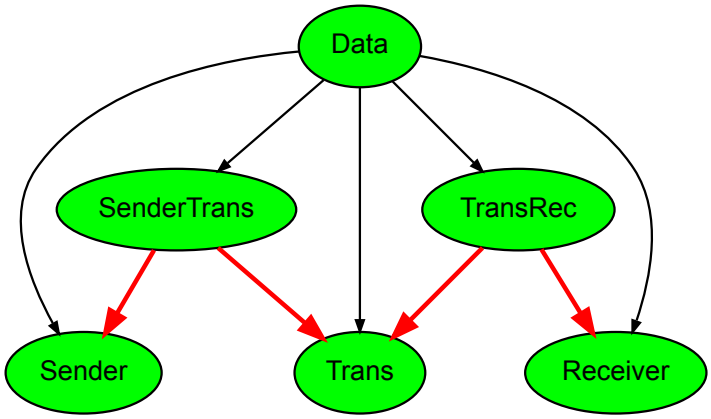
- automatic proofs
- consistency checker

commands:

- auto
- glob-decomp
- global-subsume
- loc-decomp
- local-infer
- comp
- comp-new
- cons
- hide-thm
- thm-hide
- compute-colimit
- compute-normal-form
- triangle-cons
- freeness
- zigzag-gt
- importing
- disjoint-union
- renaming
- hiding
- heterogeneity
- qualify-all-names

imported libraries:

- basic svg xml json dot symbols session pp.het pp.tex pp.xml pp.html pdf



7 Conclusion and Future Work

This thesis addressed the problem of aspect introduction and analysis in software engineering by introducing aspects in the software architecture level through graph transformation. The main contribution of this thesis is the "zigzag graph transformation." The other contribution that puts the theory into practice is the tool support implementation.

The relation between system requirements and modularization is not always straightforward. In practice, *aspects* are implemented via more than one module. The *generalized-procedure* languages do not clearly address the problem of aspects. The aspect-oriented community claims to have addressed the problem, but AOP is considered by many as a modern-day "OOP goto."

Work in *aspects* is no longer limited to the implementation phase, rather it has been moved into the earlier phases of the software development life cycle. Working with aspect introduction at the architecture level provides many benefits for documentation, product risk management, understandability, reusability, and maintainability. The aspect-oriented software development community is doing a large amount of work to *weave aspects* into models, but weaving is not a general transformation, i.e., it is not automatic.

Introducing aspects into software architectures via our zigzag graph transformation makes the system evolution mechanical. The design and analysis become comprehensive and thus clearly represented in the implementation.

The nature of aspects makes it impossible to apply any of the conventional (i.e., set-theoretic, algebraic) graph transformation approaches, since those work with exact matching. Hence, this thesis presents a special kind of matching in Section 3.4 where a single edge can match a zigzag path, explained in Section 3.3. For a given system architecture transformation rule and an application system architecture, if there exists a "zigzag matching" between the left architecture and the application architecture the *zigzag graph transformation* will automatically generate the result architecture by applying the rule on the application architecture. System architectures are diagrams, in the category of (mono) *DESC* where objects are component specifications and morphisms are specification morphisms. In all cases except that of "component addition both ways" (see Section 4.5), the construction of the result architecture is a pushout construct in the category of *SysArchsZ*.

The *properties* of a system architecture are a description of the system to be developed or an agreement on how the system should function. The category *DESC* has colimits, so generation of a system specification (*colimit*) associated with a system architecture from its component descriptions (specification) and connector specifications (morphisms) is automatic.

Besides identifying the "Zigzag homomorphisms" and elaborating on 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. In summary, this research work makes the conformance check of the new system architecture with the old architecture, the detection and resolution of conflicts, and the proof of desirable new behaviors and undesirable emergent behaviors semi-automatic.

From our example, we have experienced that proving the well-definedness and validating the properties of a system architecture is a tedious redundant job. So, we have developed some tool support that mechanized our methodology and makes our evaluation and validation process feasible. The tool is based on HETS and the MFLogic language implementation is an extension of CASL. The extended HETS tool supports the primary goal of this thesis, i.e., automatic aspect introduction through the zigzag graph transformation.

The tool support not only enables the parsing and checking the syntactical correctness of our MFLogic specification, but also allows checking the correctness of the specification in terms of static semantics. From component specifications and specification morphisms, this tool automatically generates the development graph (system architecture) and from this architecture, a single click can calculate the system specification (colimit) associated with the development graph (system architecture). The next step of this research is to extend the tool support to incorporate semi-automatic proof of property preservation or conformance checking.

Since our methodology is independent of the underlying logic, potential future work could include application of the theory in industrial settings by applying widely used architecture languages, e.g., AADL, EAST-ADL.

Bibliography

Egidio Astesiano, Michel Bidoit, Hélene Kirchner, Bernd Krieg-Brückner, Peter D Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: The common algebraic specification language. *Theoretical Computer Science*, 286(2):153–196, 2002.

Michael Barr and Charles Wells. *Category theory for computing science*, volume 49. Prentice Hall New York, 1990.

Michael Barr and Charles Wells. *Toposes, triples and theories*, volume 278. Springer Science & Business Media, 2013.

Michel Bidoit and Peter D Mosses. *CASL User Manual: Introduction to Using the Common Algebraic Specification Language*, volume 2900. Springer, 2003.

International Electrotechnical Commission et al. *Systems and software engineering: architecture description*. ISO, 2011.

Constantinos Constantinides, Therapon Skotiniotis, and Maximilian Stoerzer. AOP considered harmful. In *1st European Interactive Workshop on Aspect Systems (EIWAS)*, 2004.

Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation — Part I: Basic concepts and double pushout approach. In *Handbook of Graph Grammars*, pages 163–246, 1997.

Thomas Cottenier, Aswin Van Den Berg, and Tzilla Elrad. The motorola weavr: Model weaving in a large industrial context. *Aspect-Oriented Software Development (AOSD), Vancouver, Canada*, 32:44, 2007.

György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel 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.

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

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

Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. 1985.

Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 167–180. IEEE, 1973.

Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl J. Lieberherr, and Harold Ossher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, 2001a.

Tzilla Elrad, Robert E Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001b.

José Fiadeiro and Tom Maibaum. Towards object calculi. In *Information Systems—Correctness and Reusability, Workshop IS-CORE*, volume 91, pages 129–178, 1990.

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

José Luiz Fiadeiro and Tom Maibaum. Interconnecting formalisms: supporting modularity, reuse and incrementality. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 72–80. ACM, 1995.

JoséLuiz Fiadeiro and Tom 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.

Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. A generic approach for automatic model composition. In *International Conference on Model Driven Engineering Languages and Systems*, pages 7–15. Springer, 2007.

Robert France, Indrakshi Ray, Geri Georg, and Sudipto Ghosh. Aspect-oriented approach to early design modelling. *IEE Proceedings-Software*, 151(4):173–185, 2004.

Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry, and Sudipto 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.

David Garlan. Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 91–101. ACM, 2000.

David Garlan and Mary Shaw. *An introduction to software architecture*, volume 1. World Scientific, 1994.

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

Reiko Heckel. Graph transformation in a nutshell. *Electronic notes in theoretical computer science*, 148(1):187–198, 2006.

Ivar Jacobson and Pan-Wei Ng. *Aspect-oriented software development with use cases (addison-wesley object technology series)*. Addison-Wesley Professional, 2004.

Praveen Jayaraman, Jon Whittle, Ahmed M Elkhodary, and Hassan 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.

Wolfram Kahl. *A relation-algebraic approach to graph structure transformation*. Springer, 2002.

Wolfram Kahl. Refactoring heterogeneous relation algebras around ordered categories and converse. *J. Rel. Methods in Comp. Sci.*, 1:277–313, 2004. URL `http://www.jormics.org/`.

Gregor Kiczales. Aspect-oriented programming radical research in modularity, July-August 2006. URL `https://www.usenix.org/legacy/event/sec06/tech/slides/kiczales.pdf`.

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

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 87–98. ACM, 2009.

Jyri Laukkanen. Aspect-oriented programming, 2008.

Otávio Augusto Lazzarini Lemos, Fabiano Cutigi Ferrari, Paulo Cesar Masiero, and Cristina Videira Lopes. Testing aspect-oriented programming pointcut descriptors. In *Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 33–38. ACM, 2006.

Bass Len, Clements Paul, and Kazman Rick. Software architecture in practice. *Boston, Massachusetts Addison*, 2003.

Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, 2001.

Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1):181–224, 1993.

Brice Morin, Olivier Barais, Jean-Marc Jézéquel, and Rodrigo Ramos. Towards a generic aspect-oriented modeling framework. In *Models and Aspects workshop, at ECOOP 2007*, 2007.

Brice Morin, Olivier Barais, and Jean-Marc Jézéquel. Weaving aspect configurations for managing system variability. In *2nd International Workshop on Variability Modelling of Software-intensive Systems*, 2008a.

Brice Morin, Jacques Klein, Olivier Barais, and Jean-Marc Jézéquel. A generic weaver for supporting product lines. In *Proceedings of the 13th international workshop on Early Aspects*, pages 11–18. ACM, 2008b.

Till Mossakowski, Christian Maeder, and Klaus Lüttich. The heterogeneous tool set, Hets. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 519–522. Springer, 2007.

Till Mossakowski, Christian Maeder, Mihai Codescu, and Dominik Lucke. Hets user guide-version 0.99. *DKFI GmbH, Bremen, Germany*, 2013.

Peter D Mosses. Casl reference manual. *Lecture Notes in Computer Science*, 2960, 2004.

David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

Z. Qin, J.K. Xing, and X. Zheng. *Software Architecture*. Advanced Topics in Science and Technology in China. Springer, 2010. ISBN 9783642093746. URL `https://books.google.ca/books?id=NU-HcgAACAAJ`.

Jane Radatz, Anne Geraci, and Freny Katki. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610121990(121990):3, 1990.

Arend Rensink. The joys of graph transformation. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, 2005. URL `http://doc.utwente.nl/65184/`.

Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists*. EATCS-Monographs on Theoret. Comput. Sci. 1993. ISBN 3-540-56254-0, 0-387-56254-0. doi: 10.1007/978-3-642-77968-8.

Harold Simmons. *An introduction to category theory*. Cambridge University Press, 2011.

Clarke Siobhan and Baniassad Elisa. Aspect-oriented analysis and design: The theme approach, 2005.

I. Sommerville. *Software Engineering*. International Computer Science Series. Pearson, 2011. ISBN 9780137053469. URL `https://books.google.ca/books?id=l0egcQAACAAJ`.

Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

Jon Whittle and Praveen 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.

Jon Whittle and Praveen Jayaraman. MATA: A tool for aspect-oriented modeling based on graph transformation. In *Models in Software Engineering*, pages 16–27. Springer, 2008.

Appendix A Amalgamation Theorem and its Proof

Theorem A.0.1. *If the span* $\mathscr{A} \xleftarrow{\Xi} \mathscr{L} \xrightarrow{\Phi} \mathscr{R}$ *in SysArchsZ can be factored via three pushouts in SysArchs as shown in Figure A.0.1, and if the two SysArchsZ spans* $\mathscr{A}_1 \xleftarrow{\Xi} \mathscr{L}_1 \xrightarrow{\Phi} \mathscr{R}_1$ *and* $\mathscr{A}_2 \xleftarrow{\Xi} \mathscr{L}_2 \xrightarrow{\Phi} \mathscr{R}_2$ *have pushouts in SysArchsZ, then* $\mathscr{A} \xleftarrow{\Xi} \mathscr{L} \xrightarrow{\Phi} \mathscr{R}$ *has pushout too.*

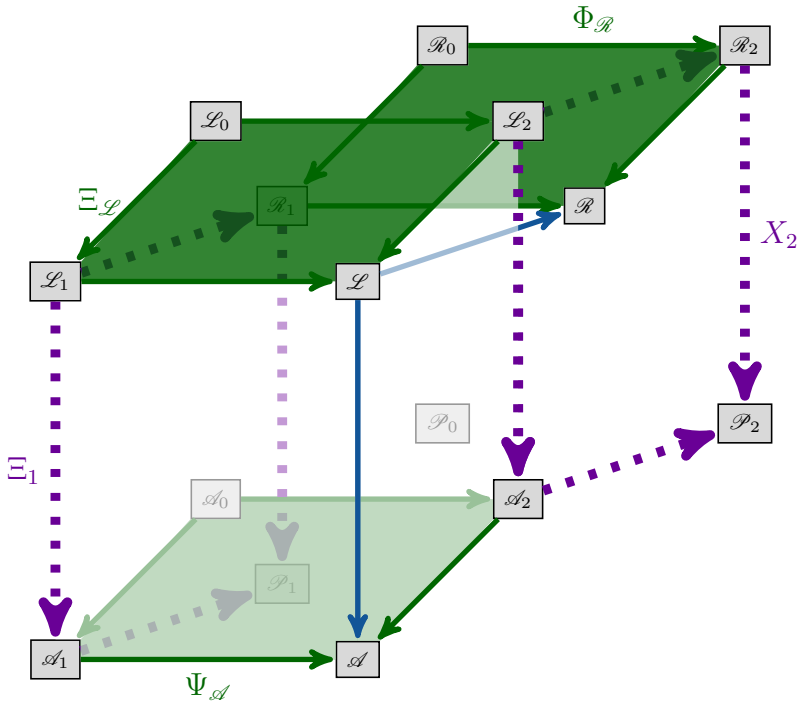


Figure A.0.1: Amalgamation Theorem: Assumption

In the above Figure,the span $\mathscr{A} \xleftarrow{\Xi} \mathscr{L} \xrightarrow{\Phi} \mathscr{R}$ is factored via three pushouts in **SysArchs** category, i.e., the $\mathscr{L}$-pushout, the $\mathscr{R}$-pushout and the $\mathscr{A}$-pushout. The $\mathscr{L}$-pushout is the pushout of the span $\mathscr{L}_1 \xleftarrow{\Xi_{\mathscr{L}}} \mathscr{L}_0 \xrightarrow{\Phi_{\mathscr{L}}} \mathscr{L}_2$, the $\mathscr{R}$-pushout is the pushout of the span $\mathscr{R}_1 \xleftarrow{\Xi_{\mathscr{R}}} \mathscr{R}_0 \xrightarrow{\Phi_{\mathscr{R}}} \mathscr{R}_2$, and the pushout of the span $\mathscr{A}_1 \xleftarrow{\Xi_{\mathscr{A}}} \mathscr{A}_0 \xrightarrow{\Phi_{\mathscr{A}}} \mathscr{A}_2$ is named as the $\mathscr{A}$-pushout. Two side pushouts are the 1-pushout and the 2-pushout, i.e., the pushout of the span $\mathscr{A}_1 \xleftarrow{\Xi_1} \mathscr{L}_1 \xrightarrow{\Phi_1} \mathscr{R}_1$ and $\mathscr{A}_2 \xleftarrow{\Xi_2} \mathscr{L}_2 \xrightarrow{\Phi_2} \mathscr{R}_2$ respectively.

The factoring makes the top square and the left-top square between the two cubes commute. They are represented in the following equation respectively.

$$\Phi_0 \ ; \ \Phi_{\mathscr{R}} = \Phi_{\mathscr{L}} \ ; \ \Phi_2 \text{ and } \Phi_0 \ ; \ \Xi_{\mathscr{R}} = \Xi_{\mathscr{L}} \ ; \ \Phi_1 \tag{3}$$

The factoring also makes the left-side square and the back square of the front cube

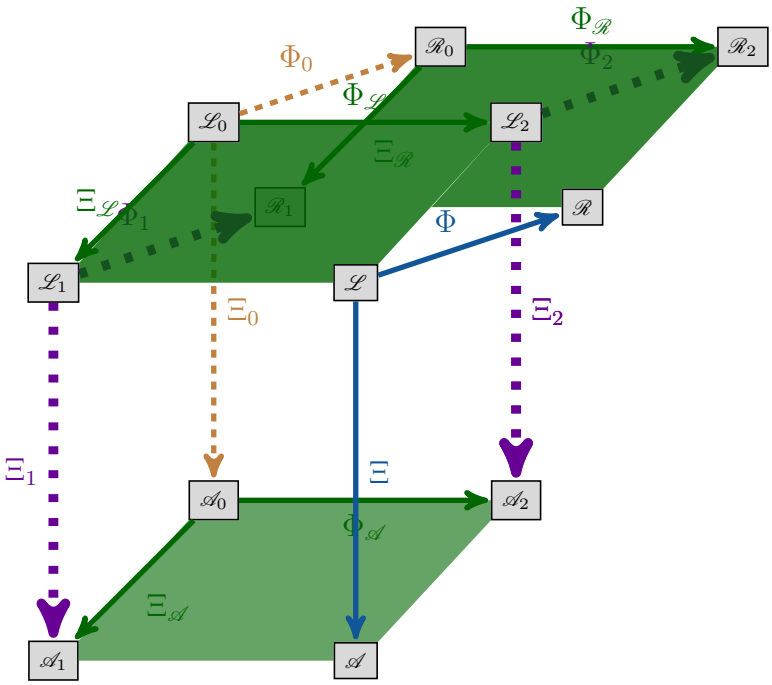


Figure A.0.2: Amalgamation Theorem: Factoring

commute. In equation:

$$\Xi_{\mathscr{L}} \; ; \; \Xi_1 = \Xi_0 \; ; \; \Xi_{\mathscr{A}} \text{ and } \Phi_{\mathscr{L}} \; ; \; \Xi_2 = \Xi_0 \; ; \; \Phi_{\mathscr{A}} \tag{4}$$

We need to prove that the span $\mathscr{A} \xleftarrow{\Xi} \mathscr{L} \xrightarrow{\Phi} \mathscr{R}$ has pushout.

Proof:

Auxiliary Part:

The system architecture $\mathscr{L}$ is factored in such a way that the underlying graph of the diagram $\mathscr{L}_0$ contains only vertices. The morphisms $\Phi_{\mathscr{L}} : \mathscr{L}_0 \to \mathscr{L}_2$ and $\Xi_{\mathscr{L}} : \mathscr{L}_0 \to \mathscr{L}_1$ are mono. The *system architectures* $\mathscr{R}$ and $\mathscr{A}$ are factored by keeping consistency with the factorization of $\mathscr{L}$. The underlying graphs of the *system architectures* $\mathscr{R}_0$, $\mathscr{A}_0$ and $\mathscr{P}_0$ also contains only vertices and $\mathscr{P}_0 := \mathscr{R}_0 \coprod_{\mathscr{L}_0} \mathscr{A}_0$. That is, in *SysArchs*, the object (*system architecture*) $\mathscr{P}_0$ along with the *system architecture* morphisms $X_0 : \mathscr{R}_0 \to \mathscr{P}_0$,$\Psi_0 : \mathscr{A}_0 \to \mathscr{P}_0$, is a pushout over the morphisms $\Phi_0 : \mathscr{L}_0 \to \mathscr{R}_0$ and $\Xi_0 : \mathscr{L}_0 \to \mathscr{A}_0$. Very similarly, the object $\mathscr{P}$ along with the *system architecture homomorphisms* $X_{\mathscr{P}} : \mathscr{P}_2 \to \mathscr{P}$,$\Psi_{\mathscr{P}} : \mathscr{P}_1 \to \mathscr{P}$, is obtained as a pushout in *SysArchs* over the morphisms $\Phi_{\mathscr{P}} : \mathscr{P}_0 \to \mathscr{P}_2$ and $\Xi_{\mathscr{P}} : \mathscr{P}_0 \to \mathscr{P}_1$. According to the theorem 4.4.1 all the pushout in *SysArchs* are pushout in *SysArchsZ*.

If $\mathscr{P}_2$ is considered as another object for the 0-pushout, where $\Phi_0 \,;\, \Phi_{\mathscr{R}} \,;\, X_2 = \Xi_0 \,;\, \Phi_{\mathscr{A}} \,;\, \Psi_2$:

$$\begin{aligned}
& \Phi_0 \,;\, \Phi_{\mathscr{R}} \,;\, X_2 \\
= \quad & \langle \text{From Equation 3, } \Phi_0 \,;\, \Phi_{\mathscr{R}} = \Phi_{\mathscr{L}} \,;\, \Phi_2 \rangle \\
& \Phi_{\mathscr{L}} \,;\, \Phi_2 \,;\, X_2 \\
= \quad & \langle \text{From 2-pushout, } \Phi_2 \,;\, X_2 = \Xi_2 \,;\, \Psi_2 \rangle \\
& \Phi_{\mathscr{L}} \,;\, \Xi_2 \,;\, \Psi_2 \\
= \quad & \langle \text{From Equation 4, } \Phi_{\mathscr{L}} \,;\, \Xi_2 = \Xi_0 \,;\, \Phi_{\mathscr{A}} \rangle \\
& \Xi_0 \,;\, \Phi_{\mathscr{A}} \,;\, \Psi_2
\end{aligned}$$

According to the definition of pushout in *SysArchsZ*, there must be a unique *system architecture zigzag homomorphism* $\Phi_{\mathscr{P}} : \mathscr{P}_0 \to \mathscr{P}_2$ such that:

$$\Phi_{\mathscr{R}} \,;\, X_2 = X_0 \,;\, \Phi_{\mathscr{P}} \text{ and } \Psi_0 \,;\, \Phi_{\mathscr{P}} = \Phi_{\mathscr{A}} \,;\, \Psi_2 \tag{5}$$

If $\mathscr{P}_1$ is considered as another object for the 0-pushout, where $\Phi_0 \,;\, \Xi_{\mathscr{R}} \,;\, X_1 = \Xi_0 \,;\, \Xi_{\mathscr{A}} \,;\, \Psi_1$:

$$\begin{aligned}
& \Phi_0 \,;\, \Xi_{\mathscr{R}} \,;\, X_1 \\
= \quad & \langle \text{From Equation 3, } \Phi_0 \,;\, \Xi_{\mathscr{R}} = \Xi_{\mathscr{L}} \,;\, \Phi_1 \rangle \\
& \Xi_{\mathscr{L}} \,;\, \Phi_1 \,;\, X_1 \\
= \quad & \langle \text{From 1-pushout, } \Phi_1 \,;\, X_1 = \Xi_1 \,;\, \Psi_1 \rangle \\
& \Xi_{\mathscr{L}} \,;\, \Xi_1 \,;\, \Psi_1 \\
= \quad & \langle \text{From Equation 4, } \Xi_{\mathscr{L}} \,;\, \Xi_1 = \Xi_0 \,;\, \Xi_{\mathscr{A}} \rangle \\
& \Xi_0 \,;\, \Xi_{\mathscr{A}} \,;\, \Psi_1
\end{aligned}$$

According to the definition of pushout in *SysArchsZ*, there must be a unique *system architecture zigzag homomorphism* $\Xi_{\mathscr{P}} : \mathscr{P}_0 \to \mathscr{P}_1$ such that:

$$\Xi_{\mathscr{R}} \,;\, X_1 = X_0 \,;\, \Xi_{\mathscr{P}} \text{ and } \Xi_{\mathscr{A}} \,;\, \Psi_1 = \Psi_0 \,;\, \Xi_{\mathscr{P}} \tag{6}$$

If $\mathscr{A}$ is considered as another object for the $\mathscr{L}$-pushout, where $\Phi_{\mathscr{L}} \,;\, \Xi_2 \,;\, X_{\mathscr{A}} = \Xi_{\mathscr{L}} \,;\, \Xi_1 \,;\, \Psi_{\mathscr{A}}$:

$$\begin{aligned}
& \Phi_{\mathscr{L}} \,;\, \Xi_2 \,;\, X_{\mathscr{A}} \\
= \quad & \langle \text{From Equation 4, } \Phi_{\mathscr{L}} \,;\, \Xi_2 = \Xi_0 \,;\, \Phi_{\mathscr{A}} \rangle \\
& \Xi_0 \,;\, \Phi_{\mathscr{A}} \,;\, X_{\mathscr{A}} \\
= \quad & \langle \text{From } \mathscr{A}\text{-pushout, } \Phi_{\mathscr{A}} \,;\, X_{\mathscr{A}} = \Xi_{\mathscr{A}} \,;\, \Psi_{\mathscr{A}} \rangle \\
& \Xi_0 \,;\, \Xi_{\mathscr{A}} \,;\, \Psi_{\mathscr{A}} \\
= \quad & \langle \text{From Equation 4, } \Xi_{\mathscr{L}} \,;\, \Xi_1 = \Xi_0 \,;\, \Xi_{\mathscr{A}} \rangle \\
& \Xi_{\mathscr{L}} \,;\, \Xi_1 \,;\, \Psi_{\mathscr{A}}
\end{aligned}$$

According to the definition of pushout in *SysArchsZ*, there must be a unique *system architecture zigzag homomorphism* $\Xi : \mathscr{L} \to \mathscr{A}$ such that:

$$X_{\mathscr{L}} \ ; \ \Xi = \Xi_2 \ ; \ X_{\mathscr{A}} \text{ and } \Xi_1 \ ; \ \Psi_{\mathscr{A}} = \Psi_{\mathscr{L}} \ ; \ \Xi \tag{7}$$

If $\mathscr{R}$ is considered as another object for the $\mathscr{L}$-pushout, where $\Phi_{\mathscr{L}} \ ; \ \Phi_2 \ ; \ X_{\mathscr{R}} = \Xi_{\mathscr{L}} \ ; \ \Phi_1 \ ; \ \Psi_{\mathscr{R}}$:

$$\begin{aligned}
& \Phi_{\mathscr{L}} \ ; \ \Phi_2 \ ; \ X_{\mathscr{R}} \\
= \quad & \langle \text{From Equation 3, } \Phi_0 \ ; \ \Phi_{\mathscr{R}} = \Phi_{\mathscr{L}} \ ; \ \Phi_2 \rangle \\
& \Phi_0 \ ; \ \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} \\
= \quad & \langle \text{From } \mathscr{R}\text{-pushout, } \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} = \Xi_{\mathscr{R}} \ ; \ \Psi_{\mathscr{R}} \rangle \\
& \Phi_0 \ ; \ \Xi_{\mathscr{R}} \ ; \ \Psi_{\mathscr{R}} \\
= \quad & \langle \text{From Equation 3, } \Phi_0 \ ; \ \Xi_{\mathscr{R}} = \Xi_{\mathscr{L}} \ ; \ \Phi_1 \rangle \\
& \Xi_{\mathscr{L}} \ ; \ \Phi_1 \ ; \ \Psi_{\mathscr{R}}
\end{aligned}$$

According to the definition of pushout in *SysArchsZ*, there must be a unique *system architecture zigzag homomorphism* $\Phi : \mathscr{L} \to \mathscr{R}$ such that:

$$\Phi_2 \ ; \ X_{\mathscr{R}} = X_{\mathscr{L}} \ ; \ \Phi \text{ and } \Phi_1 \ ; \ \Psi_{\mathscr{R}} = \Psi_{\mathscr{L}} \ ; \ \Phi \tag{8}$$

If $\mathscr{P}$ is a pushout object and $\mathscr{P}'$ is another candidate makes the outer square commutes, the universal property of the pushout can be written as follows:

$$\forall \mathscr{U}_1, \mathscr{U}_2 : \mathscr{P} \to \mathscr{P}' | X \ ; \ \mathscr{U}_i = X' \text{ and } \Psi \ ; \ \mathscr{U}_i = \Psi' \bullet \mathscr{U}_1 = \mathscr{U}_2 \tag{9}$$

Main Part:

As a first step of the proof, the following section shows that the *system architecture zigzag morphisms* $X : \mathscr{R} \to \mathscr{P}$ and $\Psi : \mathscr{A} \to \mathscr{P}$ exist, and that the goal square commutes, i.e.,

$$\Phi \ ; \ X = \Xi \ ; \ \Psi. \tag{10}$$

$X : \mathscr{R} \to \mathscr{P}$ exists:

It is given that the object $\mathscr{R}$ together with the two morphisms $X_{\mathscr{R}} : \mathscr{R}_2 \to \mathscr{R}$, $\Psi_{\mathscr{R}} : \mathscr{R}_1 \to \mathscr{R}$ is a pushout in *SysArchs* for the span $\mathscr{R}_1 \xleftarrow{\Xi_{\mathscr{R}}} \mathscr{R}_0 \xrightarrow{\Phi_{\mathscr{R}}} \mathscr{R}_2$ and therefore also a pushout in *SysArchsZ*. If the *system architecture* $\mathscr{P}$ is considered as another object where $\Phi_{\mathscr{R}} \ ; \ X_2 \ ; \ X_{\mathscr{P}} = \Xi_{\mathscr{R}} \ ; \ X_1 \ ; \ \Psi_{\mathscr{P}}$:

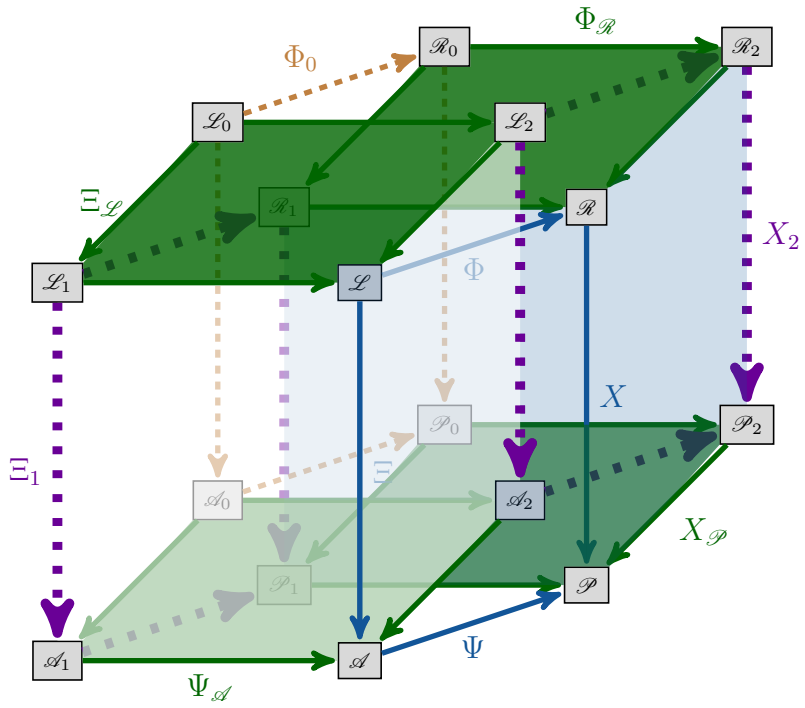


Figure A.0.3: Amalgamation Theorem: Auxiliary Part

$$
\begin{aligned}
& \Phi_{\mathscr{R}} \ ; \ X_2 \ ; \ X_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 5, } \Phi_{\mathscr{R}} \ ; \ X_2 = X_0 \ ; \ \Phi_{\mathscr{P}} \ \rangle \\
& X_0 \ ; \ \Phi_{\mathscr{P}} \ ; \ X_{\mathscr{P}} \\
= \quad & \langle \mathscr{P}\text{-pushout } \Phi_{\mathscr{P}} \ ; \ X_{\mathscr{P}} = \Xi_{\mathscr{P}} \ ; \ \Psi_{\mathscr{P}} \rangle \\
& X_0 \ ; \ \Xi_{\mathscr{P}} \ ; \ \Psi_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 6, } X_0 \ ; \ \Xi_{\mathscr{P}} = \Xi_{\mathscr{R}} \ ; \ X_1 \rangle \\
& \Xi_{\mathscr{R}} \ ; \ X_1 \ ; \ \Psi_{\mathscr{P}}
\end{aligned}
$$

According to the definition of pushout in *SysArchsZ*, there must be a unique *system architecture zigzag morphism* $X : \mathscr{R} \to \mathscr{P}$ such that

$$X_{\mathscr{R}} \ ; \ X = X_2 \ ; \ X_{\mathscr{P}} \text{ and } \Psi_{\mathscr{R}} \ ; \ X = X_1 \ ; \ \Psi_{\mathscr{P}} \tag{11}$$

$\Psi : \mathscr{A} \to \mathscr{P}$ exists:

It is also given that the object $\mathscr{A}$ along with the co-span $\mathscr{A}_1 \xrightarrow{\Psi_{\mathscr{A}}} \mathscr{A} \xleftarrow{X_{\mathscr{A}}} \mathscr{A}_2$ is a pushout in *SysArchs* over the span $\mathscr{A}_1 \xleftarrow{\Xi_{\mathscr{A}}} \mathscr{A}_0 \xrightarrow{\Phi_{\mathscr{A}}} \mathscr{A}_2$ and therefore also a pushout in *SysArchsZ*. If the *system architecture* $\mathscr{P}$ is considered as another object where $\Phi_{\mathscr{A}} \ ; \ \Psi_2 \ ; \ X_{\mathscr{P}} = \Xi_{\mathscr{A}} \ ; \ \Psi_1 \ ; \ \Psi_{\mathscr{P}}$:

$$
\begin{aligned}
& \Xi_{\mathscr{A}} \ ; \ \Psi_1 \ ; \ \Psi_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 6, } \Xi_{\mathscr{A}} \ ; \ \Psi_1 = \Psi_0 \ ; \ \Xi_{\mathscr{P}} \ \rangle \\
& \Psi_0 \ ; \ \Xi_{\mathscr{P}} \ ; \ \Psi_{\mathscr{P}} \\
= \quad & \langle \mathscr{P}\text{-pushout } \Phi_{\mathscr{P}} \ ; \ X_{\mathscr{P}} = \Xi_{\mathscr{P}} \ ; \ \Psi_{\mathscr{P}} \rangle \\
& \Psi_0 \ ; \ \Phi_{\mathscr{P}} \ ; \ X_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 5, } \Psi_0 \ ; \ \Phi_{\mathscr{P}} = \Phi_{\mathscr{A}} \ ; \ \Psi_2 \ \rangle \\
& \Phi_{\mathscr{A}} \ ; \ \Psi_2 \ ; \ X_{\mathscr{P}}
\end{aligned}
$$

According to the definition of pushout in *SysArchsZ*, there must be a unique *system architecture zigzag homomorphism* $\Psi : \mathscr{A} \to \mathscr{P}$ such that

$$\Psi_1 \ ; \ \Psi_{\mathscr{P}} = \Psi_{\mathscr{A}} \ ; \ \Psi \text{ and } \Psi_2 \ ; \ X_{\mathscr{P}} = X_{\mathscr{A}} \ ; \ \Psi. \tag{12}$$

So, it is proved that the morphisms X and Ψ exist. Now it is time to show that Equation 10 holds.

Commutativity (Equation 10):

We know that, the object $\mathscr{L}$ along with the co-span $\mathscr{L}_1 \xrightarrow{\Psi_{\mathscr{L}}} \mathscr{L} \xleftarrow{X_{\mathscr{L}}} \mathscr{L}_2$ is a pushout in *SysArchs* over the span $\mathscr{L}_1 \xleftarrow{\Xi_{\mathscr{L}}} \mathscr{L}_0 \xrightarrow{\Phi_{\mathscr{L}}} \mathscr{L}_2$. Let us consider the *system architecture* $\mathscr{P}$ as another object. Here,

$$
\begin{aligned}
& \Xi_{\mathscr{L}} \ ; \ \Xi_1 \ ; \ \Psi_1 \ ; \ \Psi_{\mathscr{P}} \\
= \quad & \langle \text{1-pushout, } \Xi_1 \ ; \ \Psi_1 = \Phi_1 \ ; \ X_1 \ \rangle \\
& \Xi_{\mathscr{L}} \ ; \ \Phi_1 \ ; \ X_1 \ ; \ \Psi_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 11, } X_1 \ ; \ \Psi_{\mathscr{P}} = \Psi_{\mathscr{R}} \ ; \ X \rangle \\
& \Xi_{\mathscr{L}} \ ; \ \Phi_1 \ ; \ \Psi_{\mathscr{R}} \ ; \ X \\
= \quad & \langle \text{From Equation 8, } \Phi_1 \ ; \ \Psi_{\mathscr{R}} = \Psi_{\mathscr{L}} \ ; \ \Phi \rangle \\
& \Xi_{\mathscr{L}} \ ; \ \Psi_{\mathscr{L}} \ ; \ \Phi \ ; \ X \\
= \quad & \langle \mathscr{L}\text{-pushout, } \Xi_{\mathscr{L}} \ ; \ \Psi_{\mathscr{L}} = \Phi_{\mathscr{L}} \ ; \ X_{\mathscr{L}} \rangle \\
& \Phi_{\mathscr{L}} \ ; \ X_{\mathscr{L}} \ ; \ \Phi \ ; \ X \\
= \quad & \langle \text{From Equation 8, } X_{\mathscr{L}} \ ; \ \Phi = \Phi_2 \ ; \ X_{\mathscr{R}} \rangle \\
& \Phi_{\mathscr{L}} \ ; \ \Phi_2 \ ; \ X_{\mathscr{R}} \ ; \ X \\
= \quad & \langle \text{From Equation 11, } X_{\mathscr{R}} \ ; \ X = X_2 \ ; \ X_{\mathscr{P}} \rangle \\
& \Phi_{\mathscr{L}} \ ; \ \Phi_2 \ ; \ X_2 \ ; \ X_{\mathscr{P}}
\end{aligned}
$$

According to the universal property of pushout, there must be a unique morphism $\Omega : \mathscr{L} \to \mathscr{P}$ such that

$$\Xi_1 \ ; \ \Psi_1 \ ; \ \Psi_{\mathscr{P}} = \Psi_{\mathscr{L}} \ ; \ \Omega \text{ and } \Phi_2 \ ; \ X_2 \ ; \ X_{\mathscr{P}} = X_{\mathscr{L}} \ ; \ \Omega. \tag{13}$$

From the above diagram it is obvious that for given two other morphisms Ξ ; Ψ and Φ ; X:

$$\Xi_1 \ ; \ \Psi_1 \ ; \ \Psi_{\mathscr{P}} = \Psi_{\mathscr{L}} \ ; \ \Xi \ ; \ \Psi \text{ and } \Phi_2 \ ; \ X_2 \ ; \ X_{\mathscr{P}} = X_{\mathscr{L}} \ ; \ \Xi \ ; \ \Psi. \tag{14}$$

$$\Xi_1 \ ; \ \Psi_1 \ ; \ \Psi_{\mathscr{P}} = \Psi_{\mathscr{L}} \ ; \ \Phi \ ; \ X \text{ and } \Phi_2 \ ; \ X_2 \ ; \ X_{\mathscr{P}} = X_{\mathscr{L}} \ ; \ \Phi \ ; \ X. \tag{15}$$

Proof of equations in Equation 14:

$$\begin{aligned}
& \Xi_1 \ ; \ \Psi_1 \ ; \ \Psi_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 12, } \Psi_1 \ ; \ \Psi_{\mathscr{P}} = \Psi_{\mathscr{A}} \ ; \ \Psi \ \rangle \\
& \Xi_1 \ ; \ \Psi_{\mathscr{A}} \ ; \ \Psi \\
= \quad & \langle \text{From Equation 7, } \Xi_1 \ ; \ \Psi_{\mathscr{A}} = \Psi_{\mathscr{L}} \ ; \ \Xi \rangle \\
& \Psi_{\mathscr{L}} \ ; \ \Xi \ ; \ \Psi
\end{aligned}$$

And

$$\begin{aligned}
& \Phi_2 \ ; \ X_2 \ ; \ X_{\mathscr{P}} \\
= \quad & \langle \text{2-pushout, } \Xi_2 \ ; \ \Psi_2 = \Phi_2 \ ; \ X_2 \ \rangle \\
& \Xi_2 \ ; \ \Psi_2 \ ; \ X_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 12, } \Psi_2 \ ; \ X_{\mathscr{P}} = X_{\mathscr{A}} \ ; \ \Psi \rangle \\
& \Xi_2 \ ; \ X_{\mathscr{A}} \ ; \ \Psi \\
= \quad & \langle \text{From Equation 7, } \Xi_2 \ ; \ X_{\mathscr{A}} = X_{\mathscr{L}} \ ; \ \Xi \rangle \\
& X_{\mathscr{L}} \ ; \ \Xi \ ; \ \Psi
\end{aligned}$$

Proof of equations in Equation 15:

$$\begin{aligned}
& \Xi_1 \ ; \ \Psi_1 \ ; \ \Psi_{\mathscr{P}} \\
= \quad & \langle \text{1-pushout, } \Xi_1 \ ; \ \Psi_1 = \Phi_1 \ ; \ X_1 \rangle \\
& \Phi_1 \ ; \ X_1 \ ; \ \Psi_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 11, } X_1 \ ; \ \Psi_{\mathscr{P}} = \Psi_{\mathscr{R}} \ ; \ X \rangle \\
& \Phi_1 \ ; \ \Psi_{\mathscr{R}} \ ; \ X \\
= \quad & \langle \text{From Equation 8, } \Phi_1 \ ; \ \Psi_{\mathscr{R}} = \Psi_{\mathscr{L}} \ ; \ \Phi \rangle \\
& \Psi_{\mathscr{L}} \ ; \ \Phi \ ; \ X
\end{aligned}$$

And

$$\begin{aligned}
& \Phi_2 \ ; \ X_2 \ ; \ X_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 11, } X_2 \ ; \ X_{\mathscr{P}} = X_{\mathscr{R}} \ ; \ X \ \rangle \\
& \Phi_2 \ ; \ X_{\mathscr{R}} \ ; \ X \\
= \quad & \langle \text{From Equation 8, } \Phi_2 \ ; \ X_{\mathscr{R}} = X_{\mathscr{L}} \ ; \ \Phi \rangle \\
& X_{\mathscr{L}} \ ; \ \Phi \ ; \ X
\end{aligned}$$

From Equation 13, Equation 14 and Equation 15 it is proved that $\Phi \ ; \ X = \Omega = \Xi \ ; \ \Psi$. Therefore, $\Phi \ ; \ X = \Xi \ ; \ \Psi$. So, it is proved that the goal square commutes, i.e., $\Phi \ ; \ X = \Xi \ ; \ \Psi$.

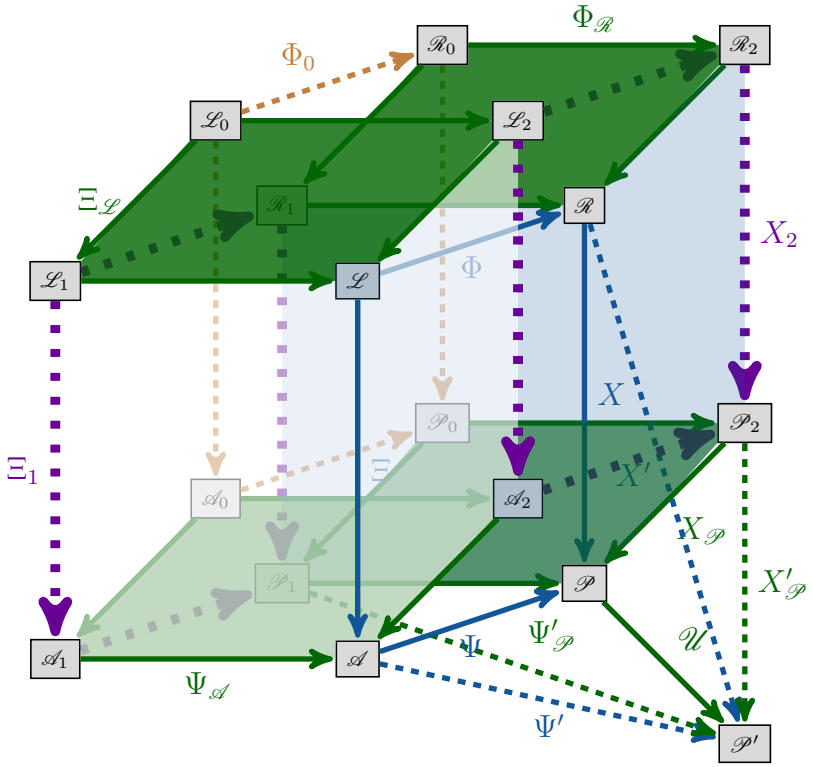


Figure A.0.4: Amalgamation Theorem: Universal Property

Universal Property of ($\mathscr{P}$, X, Ψ):

Now we need to show that ($\mathscr{P}$, X, Ψ) is universal. That is, for given another triple ($\mathscr{P}'$, $X' : \mathscr{R} \to \mathscr{P}'$, $\Psi' : \mathscr{A} \to \mathscr{P}'$), where

$$\Phi \ ; \ X' = \Xi \ ; \ \Psi' \tag{16}$$

There must be an unique arrow $\mathscr{U} : \mathscr{P} \to \mathscr{P}'$ such that

$$\Psi \ ; \ \mathscr{U} = \Psi' \text{ and } X \ ; \ \mathscr{U} = X' \tag{17}$$

For given another triple ($\mathscr{P}'$, $X' : \mathscr{R} \to \mathscr{P}'$, $\Psi' : \mathscr{A} \to \mathscr{P}'$), two other morphisms $X'_{\mathscr{P}} : \mathscr{P}_2 \to \mathscr{P}'$, $\Psi'_{\mathscr{P}} : \mathscr{P}_1 \to \mathscr{P}'$ must exist and the following equation must hold.

$$\Phi_{\mathscr{P}} \ ; \ X'_{\mathscr{P}} = \Xi_{\mathscr{P}} \ ; \ \Psi'_{\mathscr{P}} \tag{18}$$

$X'_{\mathscr{P}} : \mathscr{P}_2 \to \mathscr{P}'$ exists:

Let's consider $\mathscr{P}'$ as another object for the 2-pushout in *SysArchsZ*. Now, if it can be shown that $\Phi_2 \ ; \ X_{\mathscr{R}} \ ; \ X' = \Xi_2 \ ; \ X_{\mathscr{A}} \ ; \ \Psi'$:

$$
\begin{aligned}
& \Phi_2 \ ; \ X_{\mathscr{R}} \ ; \ X' \\
= \quad & \langle \text{From Equation 8, } \Phi_2 \ ; \ X_{\mathscr{R}} = X_{\mathscr{L}} \ ; \ \Phi \rangle \\
& X_{\mathscr{L}} \ ; \ \Phi \ ; \ X' \\
= \quad & \langle \text{From Equation 16, } \Phi \ ; \ X' = \Xi \ ; \ \Psi' \rangle \\
& X_{\mathscr{L}} \ ; \ \Xi \ ; \ \Psi' \\
= \quad & \langle \text{From Equation 7, } X_{\mathscr{L}} \ ; \ \Xi = \Xi_2 \ ; \ X_{\mathscr{A}} \rangle \\
& \Xi_2 \ ; \ X_{\mathscr{A}} \ ; \ \Psi'
\end{aligned}
$$

According to the universal property from the 2-pushout, it automatically proves that the morphism $X'_{\mathscr{P}} : \mathscr{P}_2 \to \mathscr{P}'$ exists and it is unique such that

$$X_{\mathscr{R}} \ ; \ X' = X_2 \ ; \ X'_{\mathscr{P}} \text{ and } X_{\mathscr{A}} \ ; \ \Psi' = \Psi_2 \ ; \ X'_{\mathscr{P}} \tag{19}$$

$\Psi'_{\mathscr{P}} : \mathscr{P}_1 \to \mathscr{P}'$ exists:

As we have proved the existence of $X'_{\mathscr{P}}$, very similarly for the 1-pushout and the *system architecture* $\mathscr{P}'$, it can be proved that $\Phi_1 \ ; \ \Psi_{\mathscr{R}} \ ; \ X' = \Xi_1 \ ; \ \Psi_{\mathscr{A}} \ ; \ \Psi'$. Hence, according to the universal property of the pushout, it is proved that the morphism $\Psi'_{\mathscr{P}} : \mathscr{P}_1 \to \mathscr{P}'$ exists and it is unique such that

$$\Psi_{\mathscr{A}} \ ; \ \Psi' = \Psi_1 \ ; \ \Psi'_{\mathscr{P}} \text{ and } \Psi_{\mathscr{R}} \ ; \ X' = X_1 \ ; \ \Psi'_{\mathscr{P}} \tag{20}$$

$\Phi_{\mathscr{P}} \ ; \ X'_{\mathscr{P}} = \Xi_{\mathscr{P}} \ ; \ \Psi'_{\mathscr{P}}$(*Equation 18*) :

Now, it is time to show that $\Phi_{\mathscr{P}} \ ; \ X'_{\mathscr{P}} = \Xi_{\mathscr{P}} \ ; \ \Psi'_{\mathscr{P}}$. If it can be shown that $\Phi_0 \ ; \ \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} \ ; \ X' = \Xi_0 \ ; \ \Xi_{\mathscr{A}} \ ; \ \Psi_{\mathscr{A}} \ ; \ \Psi'$:

$$
\begin{aligned}
& \Phi_0 \ ; \ \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} \ ; \ X' \\
= \quad & \langle \mathscr{R}\text{-pushout}, \ \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} = \Xi_{\mathscr{R}} \ ; \ \Psi_{\mathscr{R}} \ \rangle \\
& \Phi_0 \ ; \ \Xi_{\mathscr{R}} \ ; \ \Psi_{\mathscr{R}} \ ; \ X' \\
= \quad & \langle \text{From Equation 20}, \ \Psi_{\mathscr{R}} \ ; \ X' = X_1 \ ; \ \Psi'_{\mathscr{P}} \ \rangle \\
& \Phi_0 \ ; \ \Xi_{\mathscr{R}} \ ; \ X_1 \ ; \ \Psi'_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 3}, \ \Phi_0 \ ; \ \Xi_{\mathscr{R}} = \Xi_{\mathscr{L}} \ ; \ \Phi_1 \ \rangle \\
& \Xi_{\mathscr{L}} \ ; \ \Phi_1 \ ; \ X_1 \ ; \ \Psi'_{\mathscr{P}} \\
= \quad & \langle 1\text{-pushout}, \ \Phi_1 \ ; \ X_1 = \Xi_1 \ ; \ \Psi_1 \ \rangle \\
& \Xi_{\mathscr{L}} \ ; \ \Xi_1 \ ; \ \Psi_1 \ ; \ \Psi'_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 4}, \ \Xi_{\mathscr{L}} \ ; \ \Xi_1 = \Xi_0 \ ; \ \Xi_{\mathscr{A}} \ \rangle \\
& \Xi_0 \ ; \ \Xi_{\mathscr{A}} \ ; \ \Psi_1 \ ; \ \Psi'_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 20}, \ \Psi_1 \ ; \ \Psi'_{\mathscr{P}} = \Psi_{\mathscr{A}} \ ; \ \Psi' \ \rangle \\
& \Xi_0 \ ; \ \Xi_{\mathscr{A}} \ ; \ \Psi_{\mathscr{A}} \ ; \ \Psi'
\end{aligned}
$$

According to the universal property of pushout, there must be a unique morphism $\Omega' : \mathscr{P}_0 \to \mathscr{P}'$ such that

$$\Xi_{\mathscr{A}} \ ; \ \Psi_{\mathscr{A}} \ ; \ \Psi' = \Psi_0 \ ; \ \Omega' \text{ and } \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} \ ; \ X' = X_0 \ ; \ \Omega'. \tag{21}$$

From the above diagram it is obvious that for given two other morphisms $\Xi_{\mathscr{P}} \ ; \ \Psi'_{\mathscr{P}}$ and $\Phi_{\mathscr{P}} \ ; \ X'_{\mathscr{P}}$:

$$\Xi_{\mathscr{A}} \ ; \ \Psi_{\mathscr{A}} \ ; \ \Psi' = \Psi_0 \ ; \ \Xi_{\mathscr{P}} \ ; \ \Psi'_{\mathscr{P}} \text{ and } \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} \ ; \ X' = X_0 \ ; \ \Xi_{\mathscr{P}} \ ; \ \Psi'_{\mathscr{P}}. \tag{22}$$

$$\Xi_{\mathscr{A}} \ ; \ \Psi_{\mathscr{A}} \ ; \ \Psi' = \Psi_0 \ ; \ \Phi_{\mathscr{P}} \ ; \ X'_{\mathscr{P}} \text{ and } \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} \ ; \ X' = X_0 \ ; \ \Phi_{\mathscr{P}} \ ; \ X'_{\mathscr{P}}. \tag{23}$$

Proof of equations in Equation 22:

$$
\begin{aligned}
& \Xi_{\mathscr{A}} \ ; \ \Psi_{\mathscr{A}} \ ; \ \Psi' \\
= \quad & \langle \text{From Equation 20}, \ \Psi_{\mathscr{A}} \ ; \ \Psi' = \Psi_1 \ ; \ \Psi'_{\mathscr{P}} \ \rangle \\
& \Xi_{\mathscr{A}} \ ; \ \Psi_1 \ ; \ \Psi'_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 6}, \ \Xi_{\mathscr{A}} \ ; \ \Psi_1 = \Psi_0 \ ; \ \Xi_{\mathscr{P}} \rangle \\
& \Psi_0 \ ; \ \Xi_{\mathscr{P}} \ ; \ \Psi'_{\mathscr{P}}
\end{aligned}
$$

And

$$
\begin{aligned}
& \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} \ ; \ X' \\
= \quad & \langle \mathscr{R}\text{-pushout}, \ \Xi_{\mathscr{R}} \ ; \ \Psi_{\mathscr{R}} = \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} \ \rangle \\
& \Xi_{\mathscr{R}} \ ; \ \Psi_{\mathscr{R}} \ ; \ X' \\
= \quad & \langle \text{From Equation 20}, \ \Psi_{\mathscr{R}} \ ; \ X' = X_1 \ ; \ \Psi'_{\mathscr{P}} \ \rangle \\
& \Xi_{\mathscr{R}} \ ; \ X_1 \ ; \ \Psi'_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 6}, \ \Xi_{\mathscr{R}} \ ; \ X_1 = X_0 \ ; \ \Xi_{\mathscr{P}} \ \rangle \\
& X_0 \ ; \ \Xi_{\mathscr{P}} \ ; \ \Psi'_{\mathscr{P}}
\end{aligned}
$$

Proof of equations in Equation 23:

$$
\begin{aligned}
& \Xi_{\mathscr{A}} \ ; \ \Psi_{\mathscr{A}} \ ; \ \Psi' \\
= \quad & \langle \mathscr{A}\text{-pushout}, \ \Xi_{\mathscr{A}} \ ; \ \Psi_{\mathscr{A}} = \Phi_{\mathscr{A}} \ ; \ X_{\mathscr{A}} \ \rangle \\
& \Phi_{\mathscr{A}} \ ; \ X_{\mathscr{A}} \ ; \ \Psi' \\
= \quad & \langle \text{From Equation 19}, \ X_{\mathscr{A}} \ ; \ \Psi' = \Psi_2 \ ; \ X_{\mathscr{P}'} \rangle \\
& \Phi_{\mathscr{A}} \ ; \ \Psi_2 \ ; \ X_{\mathscr{P}'} \\
= \quad & \langle \text{From Equation 5}, \ \Phi_{\mathscr{A}} \ ; \ \Psi_2 = \Psi_0 \ ; \ \Phi_{\mathscr{P}} \rangle \\
& \Psi_0 \ ; \ \Phi_{\mathscr{P}} \ ; \ X_{\mathscr{P}'}
\end{aligned}
$$

And

$$
\begin{aligned}
& \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} \ ; \ X' \\
= \quad & \langle \text{From Equation 19}, \ X_{\mathscr{R}} \ ; \ X' = X_2 \ ; \ X'_{\mathscr{P}} \ \rangle \\
& \Phi_{\mathscr{R}} \ ; \ X_2 \ ; \ X'_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 5}, \ \Phi_{\mathscr{R}} \ ; \ X_2 = X_0 \ ; \ \Phi_{\mathscr{P}} \rangle \\
& X_0 \ ; \ \Phi_{\mathscr{P}} \ ; \ X'_{\mathscr{P}}
\end{aligned}
$$

From Equation 21, Equation 22 and Equation 23 it is proved that $\Xi_{\mathscr{P}} \ ; \ \Psi'_{\mathscr{P}} = \Omega' = \Phi_{\mathscr{P}} \ ; \ X'_{\mathscr{P}}$. Therefore, $\Xi_{\mathscr{P}} \ ; \ \Psi'_{\mathscr{P}} = \Phi_{\mathscr{P}} \ ; \ X'_{\mathscr{P}}$.
Since, it is proved that $\Phi_{\mathscr{P}} \ ; \ X'_{\mathscr{P}} = \Xi_{\mathscr{P}} \ ; \ \Psi'_{\mathscr{P}}$, according to the universal property of $\mathscr{P}$-pushout, there exists a unique morphism $\mathscr{U} : \mathscr{P} \to \mathscr{P}'$ such that

$$X_{\mathscr{P}} \ ; \ \mathscr{U} = X'_{\mathscr{P}} \text{ and } \Psi_{\mathscr{P}} \ ; \ \mathscr{U} = \Psi'_{\mathscr{P}}. \tag{24}$$

Now it is time to show that $\mathscr{U}$ also factors the triangles for the goal pushout.

$$\Psi \ ; \ \mathscr{U} = \Psi' \text{ and } X \ ; \ \mathscr{U} = X' \tag{25}$$

Let's consider $\mathscr{P}'$ as another object for $\mathscr{R}$-pushout. Now, if it can be shown that $\Phi_{\mathscr{R}} \ ; \ X_2 \ ; \ X'_{\mathscr{P}} = \Xi_{\mathscr{R}} \ ; \ X_1 \ ; \ \Psi'_{\mathscr{P}}$:

$$
\begin{aligned}
& \Phi_{\mathscr{R}} \ ; \ X_2 \ ; \ X'_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 19, } X_2 \ ; \ X'_{\mathscr{P}} = X_{\mathscr{R}} \ ; \ X' \rangle \\
& \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} \ ; \ X' \\
= \quad & \langle \mathscr{R}\text{-pushout, } \Phi_{\mathscr{R}} \ ; \ X_{\mathscr{R}} = \Xi_{\mathscr{R}} \ ; \ \Psi_{\mathscr{R}} \rangle \\
& \Xi_{\mathscr{R}} \ ; \ \Psi_{\mathscr{R}} \ ; \ X' \\
= \quad & \langle \text{From Equation 20, } \Psi_{\mathscr{R}} \ ; \ X' = X_1 \ ; \ \Psi'_{\mathscr{P}} \rangle
\end{aligned}
$$

According to the universal property of pushout, there must be a unique morphism $X' : \mathscr{R} \to \mathscr{P}'$ such that

$$X_2 \ ; \ X'_{\mathscr{P}} = X_{\mathscr{R}} \ ; \ X' \text{ and } X_1 \ ; \ \Psi'_{\mathscr{P}} = \Psi_{\mathscr{R}} \ ; \ X'. \tag{26}$$

Another morphism (composition) $X \ ; \ \mathscr{U}$ exists between $\mathscr{R}$ and $\mathscr{P}'$ where

$$X_2 \ ; \ X'_{\mathscr{P}} = X_{\mathscr{R}} \ ; \ X \ ; \ \mathscr{U} \text{ and } X_1 \ ; \ \Psi'_{\mathscr{P}} = \Psi_{\mathscr{R}} \ ; \ X \ ; \ \mathscr{U}. \tag{27}$$

$$
\begin{aligned}
& X_2 \ ; \ X'_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 24, } X'_{\mathscr{P}} = X_{\mathscr{P}} \ ; \ \mathscr{U} \rangle \\
& X_2 \ ; \ X_{\mathscr{P}} \ ; \ \mathscr{U} \\
= \quad & \langle \text{From Equation 11, } X_2 \ ; \ X_{\mathscr{P}} = X_{\mathscr{R}} \ ; \ X \rangle \\
& X_{\mathscr{R}} \ ; \ X \ ; \ \mathscr{U}
\end{aligned}
$$

And

$$
\begin{aligned}
& X_1 \ ; \ \Psi'_{\mathscr{P}} \\
= \quad & \langle \text{From Equation 24, } \Psi'_{\mathscr{P}} = \Psi_{\mathscr{P}} \ ; \ \mathscr{U} \rangle \\
& X_1 \ ; \ \Psi_{\mathscr{P}} \ ; \ \mathscr{U} \\
= \quad & \langle \text{From Equation 11, } X_1 \ ; \ \Psi_{\mathscr{P}} = \Psi_{\mathscr{R}} \ ; \ X \rangle \\
& \Psi_{\mathscr{R}} \ ; \ X \ ; \ \mathscr{U}
\end{aligned}
$$

Applying the universal property of pushout, i.e., Equation 9, from Equation 26 and Equation 27 it is proved that $X \ ; \ \mathscr{U} = X'$.

As the second half ($X \ ; \ \mathscr{U} = X'$) of Equation 25 is proved, very similarly the first half ($\Psi \ ; \ \mathscr{U} = \Psi'$) of the Equation 25 can be proved by proving $\mathscr{P}'$ as another object for $\mathscr{A}$-pushout.

$\mathscr{U}$ is Unique for goal-pushout:

Assume another morphism $\mathscr{V} : \mathscr{P} \to \mathscr{P}'$, such that

$$X \ ; \ \mathscr{V} = X' \text{ and } \Psi \ ; \ \mathscr{V} = \Psi' \tag{28}$$

The target is to prove that $\mathscr{V} = \mathscr{U}$ by showing that

$$X_{\mathscr{P}} \;;\; \mathscr{V} = X'_{\mathscr{P}} \text{ and } \Psi_{\mathscr{P}} \;;\; \mathscr{V} = \Psi'_{\mathscr{P}} \tag{29}$$

Since $\mathscr{U}$ is unique with that property due to the $\mathscr{P}$-pushout. From the above diagram Figure A.0.4 it can be proved that

$$X_{\mathscr{R}} \;;\; X' = X_2 \;;\; X_{\mathscr{P}} \;;\; \mathscr{V} \text{ and } \Psi_{\mathscr{A}} \;;\; \Psi' = \Psi_2 \;;\; X_{\mathscr{P}} \;;\; \mathscr{V} \tag{30}$$

$$\begin{aligned}
& X_{\mathscr{R}} \;;\; X' \\
= \quad & \langle \text{From Equation 28, } X \;;\; \mathscr{V} = X' \rangle \\
& X_{\mathscr{R}} \;;\; X \;;\; \mathscr{V} \\
= \quad & \langle \text{From Equation 11, } X_{\mathscr{R}} \;;\; X = X_2 \;;\; X_{\mathscr{P}} \rangle \\
& X_2 \;;\; X_{\mathscr{P}} \;;\; \mathscr{V}
\end{aligned}$$

And

$$\begin{aligned}
& \Psi_{\mathscr{A}} \;;\; \Psi' \\
= \quad & \langle \text{From Equation 28, } \Psi \;;\; \mathscr{V} = \Psi' \rangle \\
& \Psi_{\mathscr{A}} \;;\; \Psi \;;\; \mathscr{V} \\
= \quad & \langle \text{From Equation 12, } \Psi_{\mathscr{A}} \;;\; \Psi = \Psi_1 \;;\; \Psi_{\mathscr{P}} \rangle \\
& \Psi_1 \;;\; \Psi_{\mathscr{P}} \;;\; \mathscr{V}
\end{aligned}$$

Applying the universal property of pushout, i.e., Equation 9, from Equation 19 and Equation 30 it is proved that $X_{\mathscr{P}} \;;\; \mathscr{V} = X'_{\mathscr{P}}$. As the first half ($X_{\mathscr{P}} \;;\; \mathscr{V} = X'_{\mathscr{P}}$) of Equation 30 is proved, very similarly the second half ($\Psi_{\mathscr{P}} \;;\; \mathscr{V} = \Psi'_{\mathscr{P}}$) of the Equation 30 can be proved by considering the $\mathscr{P}'$ as another object for $\mathscr{P}_1$ pushout. Applying the universal property of pushout, i.e., Equation 9, from Equation 29 and Equation 24 it is proved that $\mathscr{V} = \mathscr{U}$.

In summary, we have shown that the morphisms X, Ψ exist and the goal square commutes, i.e., $\Phi \;;\; X = \Xi \;;\; \Psi$. For given morphisms X', Ψ' where $\Phi \;;\; X' = \Xi \;;\; \Psi'$, it is proved that there exist a unique arrow $\mathscr{U}$ such that $X \;;\; \mathscr{U} = X'$ and $\Psi \;;\; \mathscr{U} = \Psi'$. So, it is proved that, the span $\mathscr{A} \xleftarrow{\Xi} \mathscr{L} \xrightarrow{\Phi} \mathscr{R}$ has pushout.

Appendix B An Example: Application of Our Methodology

In this section, with a well-known example of the sender-receiver communication, we will illustrate our methodologies and approaches. At the very beginning, we have explained the scenario of the architecture followed by the informal description of the components. In subsubsection B.1.3 we have specified the components and connectors as well as proved some of its' essential properties along with the existence of morphisms. In subsection B.2 and subsection B.3 we have described the secured-unreliable and reliable-unsecured architecture. We have introduced our graph transformation rules in subsection B.4. Finally in subsection B.5 we have applied our graph transformation technique to any arbitrary complex architecture where sender-receiver communication exists.

B.1 Architecture of Unsecured-Unreliable Communication

Let us suppose that we have an architecture of the unsecured communication between a *Sender* and a *Receiver*. Here, the *Sender* sends a message to Transmitter (*Trans*), and the Transmitter transmits it to the *Receiver*. *Sender* and Transmitter (*Trans*) will communicate through a component *SendTrans*. On the other hand, Transmitter (*Trans*) and *Receiver* communicate through a component TransRec. Here we did abstraction of the messages; each message is associated with a message number. We are performing any action to a message number means that we are performing the action to the actual message associated with the message number.

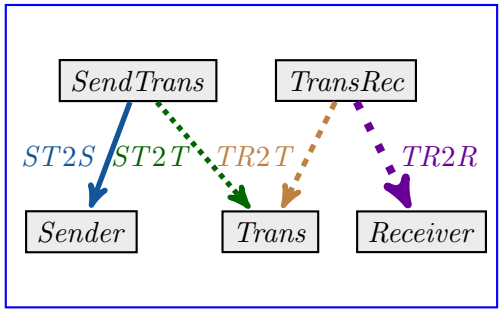


Figure B.1.1: Unsecured-Unreliable Communication Architecture

B.1.1 Scenario: Unsecured-Unreliable Communication

Type: Basic Path		
Step	Action	Component
1	Produces a message number associated with a message and sends it to the Transmitter(Trans)	Sender
2	Accepts message from Sender and transmits it to the Receiver	Trans
3	Receives the message	Receiver
1	Produces a message number associated with a message and sends it to the Transmitter(Trans)	Sender

B.1.2 Description of the components

Sender	Path Type:	Basic Path
	Objective:	Produce a message number; an abstract representation of a message and send it to a appropriate receiver, i.e., the Transmitter (Trans).
	Initial State:	Initially message number is zero and the sender is disable to send any message.
	Details Activity:	1. If the sender is not ready to send a message, it will produce a message by increasing the message number by one. It will also make the ready_to_send status enable. 2. If the ready_to_send status is enable, it will send the message to an appropriate receiver, i.e., the Transmitter (Trans) and disable the ready_to_send status.
	Final State:	Send_message attribute gets the updated message number
Transmitter	Path Type:	Basic Path
	Objective:	Access a message from an appropriate source, i.e., the Sender and transmit it to an appropriate receiver, i.e., the Receiver.
	Initial State:	Transmit_message = 0 and ready_to_transmit status is disable.

	Details Activity:	1. If the ready_to_transmit status is disable, Transmitter accept the message from appropriate source, i.e., the Sender, and switch the ready_to_transmit mode to enable. 2. Now, an action (Transmit) will be executed and the message will be transmitted to an appropriate receiver, i.e., the Receiver. The ready_to_transmit state will be set back to disable again.
	Final State:	Transmit_message will have the same value as the accepted message.
Receiver	Path Type:	Basic Path
	Objective:	Receive messages from an appropriate source, i.e., the Trans.
	Initial State:	Ready_to_receive is enable and final_message and receive_message both have initial value zero.
	Details Activity:	1. If the ready_to_receive status is enable, the Receiver receives the message, and ready_to_receive becomes disable. 2. After that, it re-state its' Ready_to_receive status to enable.
	Final State:	Final_message and receive_message attribute will have the same value; the original message sent by the Sender.

B.1.3 Formal Description of the Architecture

B.1.3.1 Component Specification

The component Sender has three attributes: *rts?* (ready to send), *msg* (message) and *send_msg* (send message). The attribute *rts?* represents whether the sender is ready to send message or not and *msg* represents the message. In order to represent the message we did abstraction and a message is represented by an integer. Two self-explanatory action symbols that are responsible for changing the state of the attributes are *Prod* and *Send*. The definition of the signature Sender is as follows:

Sender:

Sort:

Bool, *Int*

Operators:

true, *false* : *Bool*

$+$: *Int*, *Int* $\rightarrow$ *Int*

Attribute Symbol:

rts? : *Bool*

msg : *Int*

send_msg : *Int*

Action Symbol:

Send, *Prod*

Initially the *msg* is 0 and *rts?* status is false. The Sender produces a message by increasing the current value of *msg* by 1, and the *rts?* becomes true. After sending the message, *send_msg* gets a copy of the *msg*, and the *rts?* become false. The properties of the Sender can be represented by the following axioms:

S- 1: ***Beg*** $\rightarrow$ *rts? = false*

S- 2: ***Beg*** $\rightarrow$ *msg = 0*

S- 3: *(rts? = false* $\wedge$ *Prod)* $\rightarrow$ *(*$\mathbb{X}$ *rts? = true* $\wedge$ $\mathbb{X}$ *msg = msg + 1* $\wedge$ $\mathbb{X}$ *send_msg=send_msg)*

S- 4: *(rts? = true* $\wedge$ *Send)* $\rightarrow$ *(*$\mathbb{X}$ *send_msg= msg* $\wedge$ $\mathbb{X}$ *rts? = false* $\wedge$ $\mathbb{X}$ *msg=msg)*

S- 5: *(rts? = false)* $\rightarrow$ $\mathbb{F}$ *Prod*

S- 6: *(rts? = true)* $\rightarrow$ $\mathbb{F}$ *Send*

The component **Transmitter** has four attributes: *rtt?* (ready to transmit), *in_msg* (input message), *acc_msg* (accepted message) and *tra_msg* (transmitted message). These four attributes have obvious meaning: *rtt?* represents the state whether the **Transmitter** is ready to transmit the message or not, *acc_msg* is the message received from the environment and *tra_msg* is the message transmitted to an appropriate receiver. Two self-explanatory action symbols that are responsible for changing the state of attributes are *Accept* and *Transmit*. (**Transmitter** will accept message from an appropriate sender and transmit is to an appropriate receiver). The definition of the signature **Trans** is as follows:

Trans:

Sort:
Bool, *Int*
Operators:
true, *false* : *Bool*
Attribute Symbol:
rtt? : *Bool*
in_msg : *Int*
acc_msg : *Int*
tra_msg : *Int*
Action Symbol:
Accept, *Transmit*

At the very beginning the *tra_msg* is 0 and *rtt?* status is false. The **Transmitter** accept the message and *rtt?* becomes true. After transmitting the message, *tra_msg* become the message accepted by the transmitter, and *rtt?* become false.The description of the **Trans**(Transmitter) is represented by the following axioms:

T- 1: $\boldsymbol{Beg} \rightarrow rtt? = false$

T- 2: $\boldsymbol{Beg} \rightarrow tra\_msg = 0$

T- 3: $(rtt? = false \wedge Accept) \rightarrow (\mathbb{X}\, rtt? = true \wedge \mathbb{X}\, acc\_msg = in\_msg \wedge \mathbb{X}\, tra\_msg = tra\_msg)$

T- 4: $(rtt? = true \wedge Transmit) \rightarrow (\mathbb{X}\, tra\_msg = acc\_msg \wedge \mathbb{X}\, rtt? = false \wedge \mathbb{X}\, in\_msg = in\_msg)$

T- 5: $(rtt? = false) \rightarrow \mathbb{F}\, Accept$

T- 6: $(rtt? = true) \rightarrow \mathbb{F}\, Transmit$

The component **Receiver** has three attributes: *rtr?* (ready to receive), *rec_msg* (received message), and *final_msg* (final message).These thee attributes also have obvious meaning: *rtr?* represents the states whether the **Receiver** is ready to receive the message or not, *rec_msg* is the message comes as a input to the **Receiver** and *final_msg* is the message received by the **Receiver**. Two self-explanatory action symbols that are responsible for changing the state of attributes are *Receive* and *Re-state*. The definition of the signature **Receiver** is as follows:

Receiver:

Sort:
Bool, *Int*
Operators:
true, *false* : *Bool*
Attribute Symbol:
rtr? :*Bool*
rec_msg : *Int*
final_msg : *Int*
Action Symbol:
Receive, *Re−state*

Initially the *final_msg* is 0 and *rtr?* status is true. After **Receiver** receives the message from an appropriate source, the *rtr?* becomes false and *final_msg* change its' value. Finally after the execution of *Re-state* action, the attribute *rtr?* becomes true. The description of the **Receiver** is represented by the following axioms:

R- 1: $\boldsymbol{Beg} \rightarrow rtr? = true$

R- 2: $\boldsymbol{Beg} \rightarrow final\_msg = 0$

R- 3: $(rtr? = true \wedge Receive) \rightarrow (\mathbb{X}\, rtr? = false \wedge \mathbb{X}\, final\_msg = rec\_msg\ )$

R- 4: $(rtr? = false \wedge Re{-}state) \rightarrow (\mathbb{X}\, rtr? = true \wedge \mathbb{X}\, rec\_msg = rec\_msg\ )$

R- 5: $(rtr? = true) \rightarrow \mathbb{F}\, Receive$

R- 6: $(rtr? = false) \rightarrow \mathbb{F}\, Re{-}state$

The **SendTrans** is a subcomponent of **Sender** and **Trans**. Along with two other morphisms it allows the components to communicate or identify the similarities between them. The definition of the signature **SendTrans** is as follows:

SendTrans:

Sort:
Bool, *Int*
Operators:
true, *false* : *Bool*
Attribute Symbol:
sync_simsg : *Int*

Action Symbol:
SyncSendAccept

It does not have any axiom (empty).

The **TransRec** is a subcomponent of **Receiver** and **Tras**; allows them to communicate through it. The definition of the signature **TransRec** is as follows:

TransRec:

Sort:
Bool, *Int*
Operators:
true, *false* : *Bool*
Attribute Symbol:
sync_trmsg : *Int*
Action Symbol:
SyncTransRec

It does not have any axiom (empty).

B.1.3.2 Specification of Signature Morphisms and Well-definedness

ST2S: SendTrans → Sender

Sort:
Bool ↦*Bool*,
Int ↦ *Int*
Operators:
true ↦ *true*,
false ↦ *false*
Attribute Symbol:
sync_simsg ↦ *send_msg*
Action Symbol:
SyncSendAccept ↦ *Send*

Proof of well-definedness of ST2S as specification homomorphism:

In order to prove that we have a morphism between the description of **SendTrans** to **Sender** component, we have to prove that:

a The axioms of **SendTrans** are translated to theorems of **Sender** AND

b The **Sender** preserves the locality of **SendTrans**.

Proof a: Since there is no axiom, the claim **a** holds vacuously.

Proof b: We will have to prove that, $Send \vee (\mathbb{X}\, send\_msg = send\_msg)$, i.e., the translation of the locality axiom $SyncSendAccept \vee (\mathbb{X}\, sync\_simsg = sync\_simsg)$. The proof is given below:

1. $(Prod \vee Send) \vee (\mathbb{X}\, send\_msg = send\_msg \wedge \mathbb{X}\, rts? = rts? \wedge \mathbb{X}\, msg = msg)$
   locality axiom for Sender
2. $(Prod \vee Send) \vee (\mathbb{X}\, send\_msg = send\_msg)$
   1,PC
3. $Prod \rightarrow (\mathbb{X}\, rts? = true \wedge \mathbb{X}\, msg = msg+1 \wedge \mathbb{X}\, send\_msg = send\_msg)$
   axiom S-3
4. $Send \vee (\mathbb{X}\, send\_msg = send\_msg)$
   2,3, PC

ST2T: SendTrans → Trans

Sort:
$Bool \mapsto Bool$,
$Int \mapsto Int$
Operators:
$true \mapsto true$,
$false \mapsto false$
Attribute Symbol:
$sync\_simsg \mapsto in\_msg$
Action Symbol:
$SyncSendAccept \mapsto Accept$

Proof of well-definedness of ST2T as specification homomorphism:

In order to prove that we have a morphism between the description of **SendTrans** to **Trans** (Transmitter) component, we have to prove that:

a The axioms of **SendTrans** are translated to theorems of **Trans** AND

b The **Trans** preserves the locality of **SendTrans**.

Proof a: Since there is no axiom, the claim **a** holds vacuously.

Proof b: We will have to prove that, $Accept \vee (\mathbb{X}\, in\_msg = in\_msg)$, i.e., the translation of the locality axiom $SyncSendAccept \vee (\mathbb{X}\, sync\_simsg = sync\_simsg)$. The proof is given below:

1. $(Accept \vee Transmit) \vee (\mathbb{X}\, in\_msg = in\_msg \wedge \mathbb{X}\, rtt? = rtt? \wedge \mathbb{X}\, acc\_msg = acc\_msg \wedge \mathbb{X}\, tra\_msg = tra\_msg)$
   locality axiom for Sender
2. $(Accept \vee Transmit) \vee (\mathbb{X}\, in\_msg = in\_msg)$
   1,PC
3. $(rtt? = true \wedge Transmit) \rightarrow (\mathbb{X}\, tra\_msg = acc\_msg \wedge \mathbb{X}\, rtt? = false \wedge \mathbb{X}\, in\_msg = in\_msg)$
   axiom T-4
4. $Accept \vee (\mathbb{X}\, in\_msg = in\_msg)$
   2,3, PC

TR2T: TransRec → Trans

Sort:
$Bool \mapsto Bool$,
$Int \mapsto Int$
Operators:
$true \mapsto true$,
$false \mapsto false$
Attribute Symbol:
$sync\_trmsg \mapsto tra\_msg$
Action Symbol:
$SyncTransRec \mapsto Transmit$

Proof of well-definedness of TR2T as specification homomorphism:

In order to prove that we have a morphism between the description of **TransRec** to **Trans** (Transmitter) component, we have to prove that:

a The axioms of **TransRec** are translated to theorems of **Trans** AND

b The **Trans** preserves the locality of **TransRec**.

Proof a: Since there is no axiom, the claim **a** holds vacuously.

Proof b: We will have to prove that, $Transmit \vee (\mathbb{X}\, tra\_msg = tra\_msg)$, i.e., the translation of the locality axiom $SyncTransRec \vee (\mathbb{X}\, sync\_trmsg = sync\_trmsg)$. The proof is given below:

1. $(Accept \vee Transmit) \vee (\mathbb{X}\, in\_msg = in\_msg \wedge \mathbb{X}\, rtt? = rtt? \wedge \mathbb{X}\, acc\_msg = acc\_msg \wedge \mathbb{X}\, tra\_msg = tra\_msg)$
   locality axiom for Trans
2. $(Accept \vee Transmit) \vee (\mathbb{X}\, tra\_msg = tra\_msg)$
   1,PC
3. $(rtt? = false \wedge Accept) \rightarrow (\mathbb{X}\, rtt? = true \wedge \mathbb{X}\, acc\_msg = in\_msg \wedge \mathbb{X}\, tra\_msg = tra\_msg)$
   axiom T-3
4. $Transmit \vee (\mathbb{X}\, tra\_msg = tra\_msg)$
   2,3, PC

TR2R: TransRec → Receiver

Sort:
$Bool \mapsto Bool$,
$Int \mapsto Int$
Operators:
$true \mapsto true$,
$false \mapsto false$
Attribute Symbol:
$sync\_trmsg \mapsto rec\_msg$
Action Symbol:
$SyncTransRec \mapsto Receive$

Proof of well-definedness of TR2R as specification homomorphism:

In order to prove that we have a morphism between the description of **TransRec** to **Receiver** component, we have to prove that:

a The axioms of **TransRec** are translated to theorems of **Receiver** AND

b The **Receiver** preserves the locality of **TransRec**.

Proof a: Since there is no axiom, the claim **a** holds vacuously.

Proof b: We will have to prove that, *Receive* $\vee$ *(*$\mathbb{X}$ *rec_msg =rec_msg)*, i.e., the translation of the locality axiom *SyncTransRec* $\vee$ *(*$\mathbb{X}$ *sync_trmsg=sync_trmsg)*. The proof is given below:

1. *(Receive* $\vee$ *Re_state)* $\vee$ *(*$\mathbb{X}$ *rec_msg =rec_msg* $\wedge$ $\mathbb{X}$ *rtr?=rtr?* $\wedge$ $\mathbb{X}$ *final_msg=final_msg)*
   locality axiom for Sender
2. *(Receive* $\vee$ *Re_state)* $\vee$ *(*$\mathbb{X}$ *rec_msg =rec_msg)*
   1,PC
3. *(rtr? = false* $\wedge$ *Re−state)* $\rightarrow$ *(*$\mathbb{X}$ *rtr? = true* $\wedge$ $\mathbb{X}$ *rec_msg = rec_msg)*
   axiom R-4
4. *Receive* $\vee$ *(*$\mathbb{X}$ *rec_msg =rec_msg)*
   2,3, PC

B.1.4 Proof of some properties of Components

Sender

Sender $\Rightarrow$ $\neg$(Prod $\wedge$ Send)

Proof:

1. Beg	Assume
2. rts? = false	1,S-1,MP
3. (rts? = true) $\rightarrow$ Send $\vee$ NF Send	S-6,EQ
4. (rts? = true) $\rightarrow$ Send	3,Assume
5. $\neg$ Send	Assume
6. $\neg$(rts? = true)	4,5,MT
7. rts? = false	6, NA
8. $\neg$ Send 9. $\neg$ Send $\vee$ $\neg$ Prod	Addition
10. $\neg$ (Send $\wedge$ Prod)	9, DM

So, it is proved that in any state either the *Prod* or *Send* action would be executed, but not both at a time.

Assertion

Eventually a message would be produced.

Sender $\Rightarrow$ F (rts? = true $\wedge$ msg>0)

Proof:

1. Beg → rts? = false	S-1
2. Beg → msg=0	S-2
3. Beg	Assumption
4. msg=0	2,3, MP
5. rts? = false	1,3, MP
6. Prod	5,S-5,EQ,MP
7. (Xrts? = true ∧ Xmsg = msg + 1 ∧ Xsend_msg=send_msg)	5,6, MPSub
8. (Xrts? = true ∧ Xmsg = msg + 1)	7,Simp
9. (Xrts? = true ∧ Xmsg = 0 + 1)	8,Sub
10. F (rts? = true ∧ msg > 0)	9, EQ

B.1.5 Unsecured-Unreliable System Specification

UnsecUnrelSyst:

The signature of the unsecured-unreliable system **UnsecUnrealSyst** would be as follows:

Sort:

Bool, *Int*

Operators:

true, *false* : *Bool*
+ : *Int*, *Int* → *Int*

Attribute Symbol:

rts? : *Bool*
msg : *Int*
sync_simsg : *Int*
rtt? : *Bool*
acc_msg : *Int*
sync_trmsg : *Int*
rtr? : *Bool*
final_msg : *Int*

Action Symbol:

Prod
SyncSendAccept
SyncTransRec
Re−state

We can describe the unsecured-unreliable system **UnsecUnrelSyst** by the following axioms:

S- 1: ***Beg*** → *rts? = false*

S- 2: ***Beg*** → *msg = 0*

S- 3: $(rts? = false \wedge Prod) \rightarrow (\mathbb{X}\, rts? = true \wedge \mathbb{X}\, msg = msg + 1 \wedge \mathbb{X}\, sync\_simsg = sync\_msg)$

S- 4: $(rts? = true \wedge SyncSendAccept) \rightarrow (\mathbb{X}\, sync\_simsg = msg \wedge \mathbb{X}\, rts? = false \wedge \mathbb{X}\, msg = msg)$

S- 5: $(rts? = false) \rightarrow \mathbb{F}\, Prod$

S- 6: $(rts? = true) \rightarrow \mathbb{F}\, SyncSendAccept$

T- 1: $\boldsymbol{Beg} \rightarrow rtt? = false$

T- 2: $\boldsymbol{Beg} \rightarrow tra\_msg = 0$

T- 3: $(rtt? = false \wedge SyncSendAccept) \rightarrow (\mathbb{X}\, rtt? = true \wedge \mathbb{X}\, acc\_msg = sync\_simsg \wedge \mathbb{X}\, sync\_trmsg = sync\_trmsg)$

T- 4: $(rtt? = true \wedge SyncTransRec) \rightarrow (\mathbb{X}\, sync\_trmsg = acc\_msg \wedge \mathbb{X}\, rtt? = false \wedge \mathbb{X}\, sync\_msg = sync\_msg)$

T- 5: $(rtt? = false) \rightarrow \mathbb{F}\, SyncSendAccept$

T- 6: $(rtt? = true) \rightarrow \mathbb{F}\, SyncTransRec$

R- 1: $\boldsymbol{Beg} \rightarrow rtr? = true$

R- 2: $\boldsymbol{Beg} \rightarrow final\_msg = 0$

R- 3: $(rtr? = true \wedge SyncTransRec) \rightarrow (\mathbb{X}\, rtr? = false \wedge \mathbb{X}\, final\_msg = sync\_trmsg)$

R- 4: $(rtr? = false \wedge Re{-}state) \rightarrow (\mathbb{X}\, rtr? = true \wedge \mathbb{X}\, sync\_trmsg = sync\_trmsg)$

R- 5: $(rtr? = true) \rightarrow \mathbb{F}\, SyncTransRec$

R- 6: $(rtr? = false) \rightarrow \mathbb{F}\, Re{-}state$

B.2 Description of the Secured Communication Architecture

Now we want to introduce security to our given architecture. In order to make our system secure we need to add two components and two sub-components namely: Encipherer, Decipherer, SendEnci and DecRec respectively. The component Sender is synchronized with the Encipherer through sub-component SendEnci and the component Decipherer is synchronized with the Receiver through sub-component DecRec. Our new architecture looks like as follows:

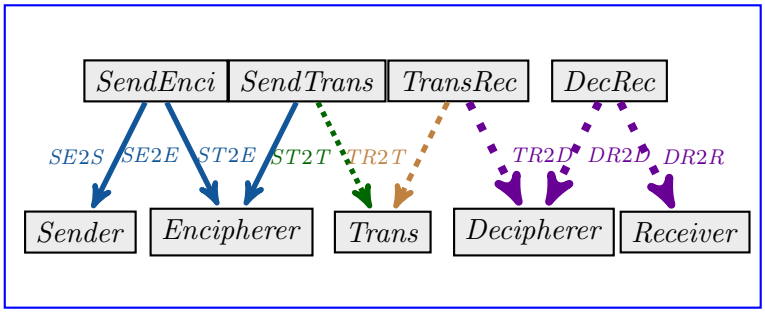


Figure B.2.1: Secured Communication Architecture

B.2.1 Scenario:Secured Communication

Type: Basic Path		
Step	Action	Component
1	Produces a message number associated to a message and sends it to Encipherer	Sender
2	Enciphers the message and sends it to Transmitter(Trans)	Encipherer
3	Transmits the message to Decipherer	Trans
4	Deciphers the message and sends it to receiver	Decipherer
5	Receives the message and re-sets its' status to receive another message	Receiver
1	Produces a message number associated to a message and sends it to Encipherer	Sender

B.2.2 Description of the New Components

Encipherer	Path Type:	Basic Path
	Initial State:	Ready_to_send status is disable and message no = 0
	Objective:	Encipher the message only and send the message number along with the encrypted message to an appropriate receiver, i.e., the Transmitter(Trans).

	Details Activity:	1. If it is not ready to send the message, it concedes the message from an appropriate source, i.e., the Sender, enciphers the message and makes its' ready_to_send status enable. 2. After that, it sends the message to an appropriate receiver, i.e., the Transmitter. Simultaneously, it disables the ready_to_send status.
	Final State:	Send_message attribute gets the message number and enciphered message associated to this message number.
Decipherer	Path Type:	Basic Path
	Initial State:	Ready_to_receive is enable and final message=0
	Objective:	Decipher encrypted message and send it to the Receiver.
	Details Activity:	1. If ready_to_receive status is enable, the Decipherer receives the message and deciphers it. At the same time, its' ready_to_receive status becomes disable. 2. After the execution of the Re-State activity, the ready_to_receive becomes enable and the message get send to the Receiver.
	Final State:	Final message is the decrypted / original message sent by the Sender.

B.2.3 Specification of the New components

The **Encipherer** has four attributes: *rts?* (ready to send), *msg* (message), *con_msg* (concede message) and *send_msg* (send message).These all attributes have obvious meaning: *rts?* represents the states whether the **Encipherer** is ready to send the message or not, *con_msg* is the message comes as a input to the **Encipherer** and *send_msg* is the message delivered to an appropriate receiver, i.e., the Transmitter (Trans). The *msg* attribute represent the enciphered message. Two self-explanatory action symbols that are responsible for changing the state of attributes are *Concede* and *Send*. The definition of the signature **Encipherer** is as follows:

Encipherer:

Sort:

Bool, *Int*

Operators:

true, *false*	:	*Bool*
+	:	*Int*, *Int* → *Int*
encipher	:	*Int* → *Int*

Attribute Symbol:

rts?	:	*Bool*
msg	:	*Int*
con_msg	:	*Int*
send_msg	:	*Int*

Action Symbol:

Send, *Concede*

Initially the *msg* is 0 and *rts?* status is false. After **Encipherer** concedes the message from an appropriate source and enciphers it, the *rts?* becomes true and *msg* change its' value. Finally after the execution of the *Send* action *rts?* becomes again false. The properties of the **Encipherer** can be represented by the following axioms:

E- 1: ***Beg*** → $rts? = false$

E- 2: ***Beg*** → $msg = 0$

E- 3: $(rts? = false \wedge Concede) \rightarrow (\mathbb{X}\, rts? = true \wedge \mathbb{X}\, msg = encipher(con\_msg) \wedge \mathbb{X}\, send\_msg = send\_msg)$

E- 4: $(rts? = true \wedge Send) \rightarrow (\mathbb{X}\, send\_msg = msg \wedge \mathbb{X}\, rts? = false \wedge \mathbb{X}\, msg = msg \wedge \mathbb{X}\, con\_msg = con\_msg)$

E- 5: $(rts? = false) \rightarrow \mathbb{F}\, Concede$

E- 6: $(rts? = true) \rightarrow \mathbb{F}\, Send$

The **Decipherer** has three attributes: *rtr?* (ready to receive), *rec_msg* (received message), and *final_msg* (final message).These thee attributes have obvious meaning: *rtr?* represents the states whether the **Decipherer** is ready to receive the message or not, *rec_msg* is the message comes as a input to the **Decipherer** and *final_msg* is the message deciphered by the **Decipherer**. Two self-explanatory action symbols that are responsible for changing the state of attributes are *Receive* and *Re-state*. The definition of the signature **Decipherer** is as follows:

Decipherer:

Sort:
Bool, *Int*
Operators:
true, *false* : *Bool*
decipher : *Int* −> *Int*
Attribute Symbol:
rtr? : *Bool*
rec_msg : *Int*
final_msg : *Int*
Action Symbol:
Receive, *Re−state*

At the very beginning the *final_msg* is 0 and *rtr?* status is true. The **Decipherer** accepts the message, deciphers it and *rtr?* becomes false. After that *Re-State* action will be executed and *rtr?* will be true. The description of the **Decipherer** is represented by the following axioms:

D- 1: ***Beg*** $\rightarrow$ *rtr? = true*

D- 2: ***Beg*** $\rightarrow$ *final_msg = 0*

D- 3: *(rtr? = true* $\wedge$ *Receive)* $\rightarrow$ *(*$\mathbb{X}$ *rtr? = false* $\wedge$ $\mathbb{X}$ *final_msg= decipher(rec_msg))*

D- 4: *(rtr? = false* $\wedge$ *Re−State)* $\rightarrow$ *(*$\mathbb{X}$ *rtr? = true* $\wedge$ $\mathbb{X}$ *rec_msg=rec_msg* $\wedge$ $\mathbb{X}$ *final_msg =final_msg)*

D- 5: *(rtr? = false)* $\rightarrow$ $\mathbb{F}$ *Re−state*

D- 6: *(rtr? = true)* $\rightarrow$ $\mathbb{F}$ *Receive*

DecRec:

Sort:
Bool, *Int*
Operators:
true, *false* : *Bool*
Attribute Symbol:
sync_frmsg : *Int*
Action Symbol:

Sync−Receive

It does not have any axiom (empty).

SendEnci:

***Sort**:*
Bool, *Int*
***Operators**:*
true, *false* : *Bool*
***Attribute Symbol**:*
sync_scmsg : *Int*
***Action Symbol**:*
SyncSendConcede

It does not have any axiom (empty).

B.2.4 Specification of New Signature Morphisms and Well-definedness

ST2E: SendTrans → Encipherer

***Sort**:*
Bool ↦*Bool*,
Int ↦ *Int*
***Operators**:*
true ↦ *true*,
false ↦ *false*
***Attribute Symbol**:*
sync_simsg ↦ *send_msg*
***Action Symbol**:*
SyncSendAccept ↦ *Send*

Proof of well-definedness of ST2E as specification homomorphism:

In order to prove that we have a morphism between the description of **SendTrans** to **Encipherer** component, we have to prove that:

a The axioms of **SendTrans** are translated to theorems of **Encipherer** AND

b The **Encipherer** preserves the locality of **SendTrans**.

Proof a: Since there is no axiom, the claim **a** holds vacuously.

Proof b: We will have to prove that, $Send \vee (\mathbb{X}\, send\_msg = send\_msg)$, i.e., the translation of the locality axiom $SyncSendAccept \vee (\mathbb{X}\, sync\_simsg = sync\_simsg)$. The proof is given below:

1. $(Concede \vee Send) \vee (\mathbb{X}\, send\_msg = send\_msg \wedge \mathbb{X}\, rts? = rts? \wedge \mathbb{X}\, msg = msg \wedge \mathbb{X}\, con\_msg = con\_msg)$
   locality axiom for Encipherer
2. $(Concede \vee Send) \vee (\mathbb{X}\, send\_msg = send\_msg)$
   1,PC
3. $Concede \rightarrow (\mathbb{X}\, rts? = true \wedge \mathbb{X}\, msg = encipher(con\_msg) \wedge \mathbb{X}\, send\_msg = send\_msg)$
   axiom E-3
4. $Send \vee (\mathbb{X}\, send\_msg = send\_msg)$
   2,3, PC

SE2S: SendEnci → Sender

Sort:
$Bool \mapsto Bool$,
$Int \mapsto Int$
Operators:
$true \mapsto true$,
$false \mapsto false$,
Attribute Symbol:
$sync\_scmsg \mapsto send\_msg$

Action Symbol:
$SyncSendConcede \mapsto Send$

Proof of well-definedness of SE2S as specification homomorphism:

In order to prove that we have a morphism between the description of **SendEnci** to **Sender** component, we have to prove that:

a The axioms of **SendEnci** are translated to theorems of **Sender** AND

b The **Sender** preserves the locality of **SendEnci**.

Proof a: Since there is no axiom, the claim **a** holds vacuously.

Proof b: We will have to prove that, $Send \vee (\mathbb{X}\, send\_msg = send\_msg)$, i.e., the translation of the locality axiom $SyncSendConcede \vee (\mathbb{X}\, sync\_scmsg = sync\_scmsg)$. The proof is given below:

1. $(Prod \vee Send) \vee (\mathbb{X}\, send\_msg = send\_msg \wedge \mathbb{X}\, rts? = rts? \wedge \mathbb{X}\, msg = msg)$
   locality axiom for Sender
2. $(Prod \vee Send) \vee (\mathbb{X}\, send\_msg = send\_msg)$
   1,PC
3. $Prod \rightarrow (\mathbb{X}\, rts? = true \wedge \mathbb{X}\, msg = msg + 1 \wedge \mathbb{X}\, send\_msg = send\_msg)$
   axiom S-3
4. $Send \vee (\mathbb{X}\, send\_msg = send\_msg)$
   2,3, PC

SE2E: SendEnci → Encipherer

Sort:
$Bool \mapsto Bool$,
$Int \mapsto Int$
Operators:
$true \mapsto true$,
$false \mapsto false$,
Attribute Symbol:
$sync\_scmsg \mapsto con\_msg$
Action Symbol:
$SyncSendConcede \mapsto Concede$

Proof of well-definedness of SE2E as specification homomorphism:

In order to prove that we have a morphism between the description of **SendEnci** to **Encipherer** component, we have to prove that:

a The axioms of **SendEnci** are translated to theorems of **Encipherer** AND

b The **Encipherer** preserves the locality of **SendEnci**.

Proof a: Since there is no axiom, the claim **a** holds vacuously.

Proof b: We will have to prove that, *Concede* $\vee$ *(*$\mathbb{X}$ *con_msg =con_msg)*, i.e., the translation of the locality axiom *SyncSendConcede* $\vee$ *(*$\mathbb{X}$ *sync_scmsg= sync_scmsg).* The proof is given below:

1. *(Concede* $\vee$ *Send)* $\vee$ *(*$\mathbb{X}$ *send_msg =send_msg* $\wedge$ $\mathbb{X}$ *rts?=rts?* $\wedge$ $\mathbb{X}$ *msg=msg* $\wedge$ $\mathbb{X}$ *con_msg=con_msg)*
   locality axiom for Encipherer
2. *(Concede* $\vee$ *Send)* $\vee$ *(*$\mathbb{X}$ *con_msg=con_msg)*
   1,PC
3. *Send* $\rightarrow$ *(*$\mathbb{X}$ *rts?=false* $\wedge$ $\mathbb{X}$ *msg=msg* $\wedge$ $\mathbb{X}$ *send_msg=msg* $\wedge$ $\mathbb{X}$ *con_msg = con_msg)*
   axiom E-4
4. *Concede* $\vee$ *(*$\mathbb{X}$ *con_msg = con_msg)*
   2,3, PC

TR2D: TransRec $\rightarrow$ Decipherer

Sort*:*
Bool $\mapsto$*Bool,*
Int $\mapsto$ *Int*
Operators*:*
true $\mapsto$ *true,*
false $\mapsto$ *false*
Attribute Symbol*:*
sync_trmsg $\mapsto$ *rec_msg*
Action Symbol*:*
SyncTransRec $\mapsto$ *Receive*

Proof of well-definedness of TR2D as specification homomorphism:

In order to prove that we have a morphism between the description of **TransRec** to **Decipherer** component, we have to prove that:

a The axioms of **TransRec** are translated to theorems of **Decipherer** AND

b The **Decipherer** preserves the locality of **TransRec**.

Proof a: Since there is no axiom, the claim **a** holds vacuously.

Proof b: We will have to prove that, *Receive* $\vee$ *(*$\mathbb{X}$ *rec_msg =rec_msg)*, i.e., the translation of the locality axiom *SyncTransRec* $\vee$ *(*$\mathbb{X}$ *sync_trmsg= sync_trmsg)*. The proof is given below:

1. *(Receive* $\vee$ *Re−state)* $\vee$ *(*$\mathbb{X}$ *final_msg =final_msg* $\wedge$ $\mathbb{X}$ *rtr?=rtr?* $\wedge$ $\mathbb{X}$ *rec_msg=rec_msg)*
   locality axiom for Decipherer
2. *(Receive* $\vee$ *Re−state)* $\vee$ *(*$\mathbb{X}$ *rec_msg=rec_msg)*
   1,PC
3. *Re−state* $\rightarrow$ *(*$\mathbb{X}$ *rtr?=true* $\wedge$ $\mathbb{X}$ *rec_msg=rec_msg* $\wedge$ $\mathbb{X}$ *final_msg = final_msg)*
   axiom D-4
4. *Receive* $\vee$ *(*$\mathbb{X}$ *rec_msg = rec_msg)*
   2,3, PC

DR2D: DecRec $\rightarrow$ Decipherer

***Sort**:*
Bool $\mapsto$*Bool,*
Int $\mapsto$ *Int*
***Operators**:*
true $\mapsto$ *true,*
false $\mapsto$ *false*
***Attribute Symbol**:*
sync_frmsg $\mapsto$ *final_msg*
***Action Symbol**:*
Sync−Receive $\mapsto$ *Receive*

Proof of well-definedness of DR2D as specification homomorphism:

In order to prove that we have a morphism between the description of **DecRec** to **Decipherer** component, we have to prove that:

a The axioms of **DecRec** are translated to theorems of **Decipherer** AND

b The **Decipherer** preserves the locality of **DecRec**.

Proof a: Since there is no axiom, the claim **a** holds vacuously.

Proof b: We will have to prove that, *Receive* $\vee$ *(*$\mathbb{X}$ *final_msg =final_msg)*, i.e., the translation of the locality axiom *Sync−Receive* $\vee$ *(*$\mathbb{X}$ *sync_frmsg= sync_frmsg)*. The proof is given below:

1. $(Receive \vee Re{-}state) \vee (\mathbb{X}\, final\_msg = final\_msg \wedge \mathbb{X}\, rtr? = rtr? \wedge \mathbb{X}\, rec\_msg = rec\_msg)$
   locality axiom for Decipherer
2. $(Receive \vee Re{-}state) \vee (\mathbb{X}\, final\_msg = final\_msg)$
   1,PC
3. $Re{-}state \rightarrow (\mathbb{X}\, rtr? = true \wedge \mathbb{X}\, rec\_msg = rec\_msg \wedge \mathbb{X}\, final\_msg = final\_msg)$
   axiom D-4
4. $Receive \vee (\mathbb{X}\, final\_msg = final\_msg)$
   2,3, PC

DR2R: DecRec → Receiver

Sort:
$Bool \mapsto Bool$,
$Int \mapsto Int$
Operators:
$true \mapsto true$,
$false \mapsto false$
Attribute Symbol:
$sync\_frmsg \mapsto rec\_msg$
Action Symbol:
$Sync{-}Receive \mapsto Receive$

Proof of well-definedness of DR2R as specification homomorphism:

In order to prove that we have a morphism between the description of **DecRec** to **Receiver** component, we have to prove that:

a The axioms of **DecRec** are translated to theorems of **Receiver** AND

b The **Receiver** preserves the locality of **DecRec**.

Proof a: Since there is no axiom, the claim **a** holds vacuously.

Proof b: We will have to prove that, $Receive \vee (\mathbb{X}\, rec\_msg = rec\_msg)$, i.e., the translation of the locality axiom $Sync{-}Receive \vee (\mathbb{X}\, sync\_frmsg = sync\_frmsg)$. The proof is given below:

1. $(Receive \vee Re{-}state) \vee (\mathbb{X}\, final\_msg = final\_msg \wedge \mathbb{X}\, rtr? = rtr? \wedge \mathbb{X}\, rec\_msg = rec\_msg)$
   locality axiom for Receiver
2. $(Receive \vee Re{-}state) \vee (\mathbb{X}\, rec\_msg = rec\_msg)$
   1,PC
3. $Re{-}state \rightarrow (\mathbb{X}\, rtr? = true \wedge \mathbb{X}\, rec\_msg = rec\_msg)$
   axiom R-4
4. $Receive \vee (\mathbb{X}\, rec\_msg = rec\_msg)$
   2,3, PC

B.2.5 Secured System Specification

SecUnrelSyst:

The signature of secured-unreliable system **SecUnrealSyst** would be as follows:

Sort:

$Bool,\ Int$

Operators:

$true,\ false$	$:$	$Bool$
$+$	$:$	$Int,\ Int \rightarrow Int$
$encipher$	$:$	$Int \rightarrow Int$
$decipher$	$:$	$Int \rightarrow Int$

Attribute Symbol:

$rts?$	$:$	$Bool$
msg	$:$	Int
$sync\_scmsg$	$:$	Int
$rts\_E?$	$:$	$Bool$
msg_E	$:$	Int
$sync\_simsg$	$:$	Int
$rtt?$	$:$	$Bool$
acc_msg	$:$	Int
$sync\_trmsg$	$:$	Int
$rtr?$	$:$	$Bool$
$sync\_frmsg$	$:$	Int
$rtr\_R?$	$:$	$Bool$
$final\_msg$	$:$	Int

Action Symbol:

$Prod$

SyncSendConcede
SyncSendAccept
Sync−Receive
Re−state
Re−state_R

We can describe the secured-unreliable system **SecUnrelSyst** by the following axioms:

R- 1: $\boldsymbol{Beg} \rightarrow rtr\_R? = true$

R- 2: $\boldsymbol{Beg} \rightarrow final\_msg = 0$

R- 3: $(rtr\_R? = true \wedge Sync{-}Receive) \rightarrow (\mathbb{X}\, rtr\_R? = false \wedge \mathbb{X}\, final\_msg = sync\_frmsg)$

R- 4: $(rtr\_R? = false \wedge Re{-}state\backslash\_R) \rightarrow (\mathbb{X}\, rtr\_R? = true \wedge \mathbb{X}\, sync\_frmsg = sync\_frmsg)$

R- 5: $(rtr\_R? = true) \rightarrow \mathbb{F}\, Sync{-}Receive$

R- 6: $(rtr\_R? = false) \rightarrow \mathbb{F}\, Re{-}state\_R$

D- 1: $\boldsymbol{Beg} \rightarrow rtr? = true$

D- 2: $\boldsymbol{Beg} \rightarrow final\_msg = 0$

D- 3: $(rtr? = true \wedge Sync{-}Receive) \rightarrow (\mathbb{X}\, rtr? = false \wedge \mathbb{X}\, sync\_frmsg = decipher(sync\_trmsg))$

D- 4: $(rtr? = false \wedge Re{-}State) \rightarrow (\mathbb{X}\, rtr? = true \wedge \mathbb{X}\, sync\_trmsg{=}sync\_trmsg \wedge \mathbb{X}\, sync\_frmsg{=}sync\_frmsg)$

D- 5: $(rtr? = false) \rightarrow \mathbb{F}\, Re{-}state$

D- 6: $(rtr? = true) \rightarrow \mathbb{F}\, Sync{-}Receive$

T- 1: $\boldsymbol{Beg} \rightarrow rtt? = false$

T- 2: $\boldsymbol{Beg} \rightarrow tra\_msg = 0$

T- 3: $(rtt? = false \wedge SyncSendAccept) \rightarrow (\mathbb{X}\, rtt? = true \wedge \mathbb{X}\, acc\_msg = sync\_simsg \wedge \mathbb{X}\, sync\_trmsg{=}sync\_trmsg)$

T- 4: $(rtt? = true \wedge Sync{-}Receive) \rightarrow (\mathbb{X}\, sync\_trmsg{=}acc\_msg \wedge \mathbb{X}\, rtt? = false \wedge \mathbb{X}\, sync\_msg{=}sync\_msg)$

T- 5: $(rtt? = false) \rightarrow \mathbb{F}\, SyncSendAccept$

T- 6: $(rtt? = true) \rightarrow \mathbb{F}\, Sync{-}Receive$

E- 1: $\boldsymbol{Beg} \rightarrow rts\_E? = false$

E- 2: $\boldsymbol{Beg} \rightarrow msg\_E = 0$

E- 3: $(rts\_E? = false \wedge SyncSendConcede) \rightarrow (\mathbb{X}\, rts\_E? = true \wedge \mathbb{X}\, msg\_E = encipher(sync\_scmsg) \wedge \mathbb{X}\, sync\_simsg{=}sync\_simsg)$

E- 4: $(rts\_E? = true \wedge SyncSendAccept) \rightarrow (\mathbb{X}\, sync\_simsg{=} msg\_E \wedge \mathbb{X}\, rts\_E? = false \wedge \mathbb{X}\, msg\_E{=}msg\_E \wedge \mathbb{X}\, sync\_scmsg{=}sync\_scmsg)$

E- 5: $(rts\_E? = false) \rightarrow \mathbb{F}\, SyncSendConcede$

E- 6: $(rts\_E? = true) \rightarrow \mathbb{F}\, SyncSendAccept$

S- 1: $\boldsymbol{Beg} \rightarrow rts? = false$

S- 2: $\boldsymbol{Beg} \rightarrow msg = 0$

S- 3: $(rts? = false \wedge Prod) \rightarrow (\mathbb{X}\, rts? = true \wedge \mathbb{X}\, msg = msg + 1 \wedge \mathbb{X}\, sync\_scmsg{=} sync\_scmsg)$

S- 4: $(rts? = true \wedge SyncSendConcede) \rightarrow (\mathbb{X}\, sync\_scmsg{=} msg \wedge \mathbb{X}\, rts? = false \wedge \mathbb{X}\, msg{=}msg)$

S- 5: $(rts? = false) \rightarrow \mathbb{F}\, Prod$

S- 6: $(rts? = true) \rightarrow \mathbb{F}\, SyncSendConcede$

B.3 Description of the Reliable Communication Architecture

Now we want to introduce reliability to our given architecture. In order to make our system reliable we need to add a couple of components namely Sender_Res, TransPlus, and Monitor. In order to synchronize with other components we need to define Trans-Montr, and MontrSend_Res component as well. Our new architecture looks like as follows:

B.3.1 Scenario:Reliable Communication

Type: Basic Path		
1	Produces message no and sends message	Sender
2	Produces and sends message by extending Sender	Sender_Res

3	Transmits all kinds of messages	Trans	
4	Transmits error free messages to the Receiver and the Monitor OR Executes step 4 of Exception Path	TransPlus	
5	Branch B1		
	Action	Step	Component
	Receives error free messages	B1-a	Receiver
	Branch B2		
	Action	Step	Component
	Monitors expected message	B2-a	Monitor
1	Produces message no and sends message	Sender	

Type: Exception Path		
Step	Action	Component
1	Produces message no and sends message	Sender
2	Produces and sends message by extending Sender	Sender_Res
3	Transmits all kinds of messages	Trans
4	Transmits error-prone messages to the Monitor OR Executes step 5 of the Basic Path	TransPlus
5	Monitors unexpected messages and sends reference to Sender_Res	Monitor
2	Re-sends lost messages	Sender_Res

B.3.2 Description of the New Components

Sender_Res	Path Type:	Basic Path
	Initial State:	Initially message No = 0 and the sender is disable to send any message.
	Objective:	Produce and send messages by extending Sender.

	Details Activity:	1. If **Sender** is not ready to send a message, it receive /produce a message (Speed) from an appropriate source, i.e., the SpeedSensor of a vehicle and increase the message number by one. It also makes the ready_to_send status enable. 2. If ready_to_send status is enable, it sends the message number and associated message to an appropriate receiver, i.e., the Transmitter(Trans) and disables the ready_to_send status
	Final State:	Send_message attribute gets the updated message number and the message associated to this number.
Sender_Res	Path Type:	Exception Path
	Initial State:	Initially lost_message = 0.
	Objective:	Resend lost messages by extending Sender.
	Details Activity:	1. If ready_to_send it not enable, and lost_message in not equal to zero, it produces the lost message (Speed). It also makes the ready to send status enable. 2. If ready_to_send status is enable, it sends the message number and associated message to an appropriate receiver,i.e., the Transmitter(Trans) and disables the ready_to_send status
	Final State:	Send message attribute gets the lost message number and the message associated to this number.
TransPlus	Path Type:	Basic Path
	Initial State:	Ready_to_transmit, ready_to_transmit_all both attributes are disable and transmit_message has value zero.
	Objective:	Transmit error free messages by extending Trans.

	Details Activity:	1. If ready_to_transmit and transmit_all status are disable, Transmitter accepts the message from appropriate source, i.e., the Sender_Res and switch the ready_to_transmit and transmit_all mode to enable. 2. Now, Transmit action will be executed and message will be transmitted to an appropriate receiver, i.e., the Receiver. The ready_to_transmit state will be set back to disable. 3. Simultaneously, Transmit_all action will be executed and message will be transmitted to an appropriate receiver, i.e., the Monitor. The ready_to_transmit_all state will be set back to disable.
	Final State:	Transmit_message attribute will have error free message number and the message associated to this number.
TransPlus	Path Type:	Exception Path
	Initial State:	Ready_to_transmit, ready_to_transmit_all both attributes are disable and transmit message has value zero.
	Objective:	Transmit error-prone messages by extending Trans.

	Details Activity:	1. If ready_to_transmit and transmit_all status are disable, Transmitter accept the message from appropriate source, i.e., the Sender_Res and switch the ready_to_transmit and transmit_all mode to enable. 2. Since the message is not error free, no message would be transmitted to its' destination, i.e., the Receiver. 3. Now, Transmit_all action will be executed and message will be transmitted to an appropriate receiver, i.e., the Monitor. The ready_to_transmit all state will be set back to disable.
	Final State:	Transmit_message attribute will have error-prone message number and the message associated to this number.
Monitor	Path Type:	Basic Path
	Initial State:	Ready_to_monitor status is enable and received_message and expected_message both are initialized to zero.
	Objective:	Monitor expected messages.
	Details Activity:	1. If ready_to_monitor is enable and received expected message from an appropriate source, i.e., the TransPlus, it will increase the expected message number by one.
	Final State:	Expected message number will have the up to date value.
Monitor	Path Type:	Exception Path
	Initial State:	Ready_to_monitor status is enable and received_message and expected_message both are initialized to zero.
	Objective:	Monitor unexpected messages.

	Details Activity:	1. If ready_to_monitor is enable and the message received is not the expected one, the expected message will remain as it was and ready_to_monitor will become disable. 2. If ready_to_monitor is disable, monitor will send a request to an appropriate sender, i.e., the Sender_Res to resend the lost message by sending the expected message number. It will also change the monitor status to enable.
	Final State:	retransmit message will have the expected message number.

B.3.3 Specification of the New components

Sender_Res:

Sender is a sub-component of Sender_Res. The Sender_Res has the ability to reproduce the lost messages and send it for transmission.

Sort:
Bool, *Int*

Operators:
true, *false* : *Bool*
+ : *Int*, *Int* → *Int*
notZero : *Int* → *Bool*

Attribute Symbol:
rts? :*Bool*
msg : *Int*
send_msg : *Int*
lost_msg : *Int*

Action Symbol:
Send, Prod,Re_Prod

The properties of the **Sender_Res** can be represented by the following axioms:

S-R 1: ***Beg*** → *rts? = false*

S-R 2: ***Beg*** → *msg = 0*

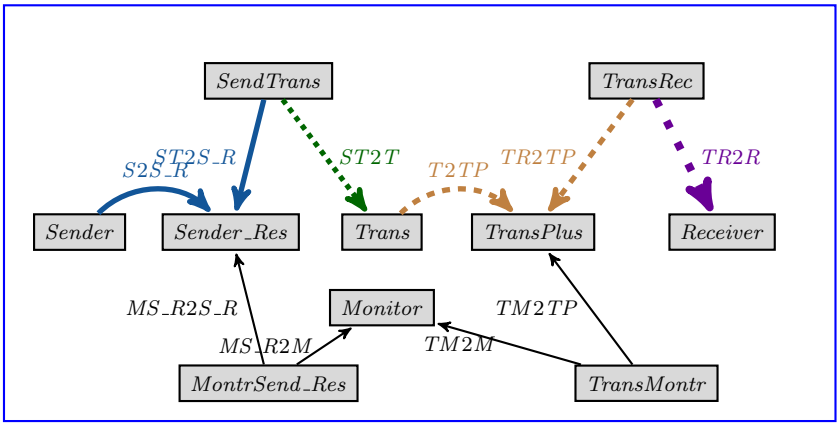


Figure B.3.1: Reliable Communication Architecture

S-R 3: ***Beg*** $\rightarrow$ *lost_msg = 0*

S-R 4: *(rts? = false* $\wedge$ *notZero(lost_msg)=false* $\wedge$ *Prod)* $\rightarrow$ $\mathbb{X}$ *rts? = true* $\wedge$ $\mathbb{X}$ *msg = msg + 1* $\wedge$ $\mathbb{X}$ *send_msg=send_msg* $\wedge$ $\mathbb{X}$ *lost_msg=lost_msg) (rts? = false* $\wedge$ *notZero (lost_msg)=true* $\wedge$ *Re_Prod)* $\rightarrow$ *(*$\mathbb{X}$ *rts? = true* $\wedge$ $\mathbb{X}$ *msg = lost_msg* $\wedge$ $\mathbb{X}$ *send_msg =send_msg* $\wedge$ $\mathbb{X}$ *lost_msg = 0)*

S-R 5: *(rts? = true* $\wedge$ *Send)* $\rightarrow$ *(*$\mathbb{X}$ *send_msg= msg* $\wedge$ $\mathbb{X}$ *rts? = false* $\wedge$ $\mathbb{X}$ *msg=msg* $\wedge$ $\mathbb{X}$ *lost_msg=lost_msg)*

S-R 6: *(rts? = false)* $\rightarrow$ $\mathbb{F}$ *Prod*

S-R 7: *(rts? = false)* $\rightarrow$ $\mathbb{F}$ *Re_Prod*

S-R 8: *(rts? = true)* $\rightarrow$ $\mathbb{F}$ *Send*

TransPlus:

Sort:
Bool, *Int*
Operators:
true, *false* : *Bool*
isEfree : *Int* $\rightarrow$ *Bool*
Attribute Symbol:
rtt? : *Bool*
in_msg : *Int*
acc_msg : *Int*
tra_msg : *Int*

tra_msg_m : *Int*

Action Symbol:

Accept, *Transmit*, *Transmit_M*

The description of the **Transmitter Plus** is represented by the following examples:

TP- 1: ***Beg*** $\rightarrow$ *rtt? = false*

TP- 2: ***Beg*** $\rightarrow$ *tra_msg_m = 0*

TP- 3: ***Beg*** $\rightarrow$ *tra_msg = 0*

TP- 4: *(rtt ? = false* $\wedge$ *Accept)* $\rightarrow$ *(*$\mathbb{X}$ *rtt? = true* $\wedge$ $\mathbb{X}$ *acc_msg = in_msg* $\wedge$ $\mathbb{X}$ *tra_msg= tra_msg)*

TP- 5: *(rtt ? = true* $\wedge$ *isEfree (acc_msg) = true* $\wedge$ *Transmit)* $\rightarrow$ *(*$\mathbb{X}$ *tra_msg=acc_msg* $\wedge$ $\mathbb{X}$ *rtt? = false* $\wedge$ $\mathbb{X}$ *in_msg=in_msg)*

TP- 6: *(rtt ? = true* $\wedge$ *isEfree (acc_msg) = false)* $\rightarrow$ *(*$\neg$ $\mathbb{F}$ *Transmit)*

TP- 7: *(rtt ? = true* $\wedge$ *Transmit_M)* $\rightarrow$ *(*$\mathbb{X}$ *tra_msg_m=acc_msg* $\wedge$ $\mathbb{X}$ *in_msg=in_msg* $\wedge$ $\mathbb{X}$ *tra_msg=tra_msg)*

TP- 8: *(rtt ? = false)* $\rightarrow$ $\mathbb{F}$ *Accept*

TP- 9: *(rtt ? = true)* $\rightarrow$ $\mathbb{F}$ *Transmit_M*

TP- 10: *(rtt ? = true* $\wedge$ *isEfree (acc_msg) = true)* $\rightarrow$ $\mathbb{F}$ *Transmit*

Monitor:

Sort:

Bool, *Int*

Operators:

true, *false* : *Bool*

+ : *Int*, *Int* $\rightarrow$ *Int*

isExpMessage: *Int*, *Int* $\rightarrow$ *Bool*

Attribute Symbol:

rtm? : *Bool*

in_msg : *Int*

rst_msg : *Int*

exp_msg : *Int*

Action Symbol:

Accept, *Resend_Req*

The description of the **Monitor** is represented by the following examples:

M- 1: ***Beg*** $\rightarrow$ *rtm? = true*

M- 2: ***Beg*** $\rightarrow$ *rst_msg = 0*

M- 3: ***Beg*** $\rightarrow$ *exp_msg= 0*

M- 4: *(rtm? = true* $\wedge$ *isExpMessage(exp_msg,in_msg)= true* $\wedge$ *Accept)* $\rightarrow$ *(*$\mathbb{X}$ *exp_msg = exp_msg + 1)*

M- 5: *(rtm? = true* $\wedge$ *isExpMessage(exp_msg,in_msg)= false* $\wedge$ *Accept)* $\rightarrow$ *(*$\mathbb{X}$ *exp_msg = exp_msg* $\wedge$ *rtm?= false)*

M- 6: *(rtm? = false* $\wedge$ *Resend_Req)* $\rightarrow$ *(* $\mathbb{X}$ *rst_msg=exp_msg* $\wedge$ *rtm?= true)*

M- 7: *(rtm? = true)* $\rightarrow$ $\mathbb{F}$ *Accept*

M- 8: *(rtm? = false)* $\rightarrow$ $\mathbb{F}$ *Resend_Req*

TransMontr:

Sort:
Bool, *Int*
Operators:
true, *false* : *Bool*
Attribute Symbol:
sync_timsg : *Int*
Action Symbol:
SyncTrans(T)Accept(M)

It does not have any axiom (empty).

MontrSend_Res:

Sort:
Bool, *Int*
Operators:
true, *false* : *Bool*
Attribute Symbol:
sync_rlmsg : *Int*
Action Symbol:
SyncResendProd

It does not have any axiom (empty).

B.3.4 Specification of New Signature Morphisms and Well-definedness

The specification (signature) of the different signature morphisms are as follows:

S2S_R: Sender → Sender_Res

Sort:
Bool ↦*Bool*,
Int ↦ *Int*
Operators:
true ↦ *true*,
false ↦ *false*
+ ↦ *+*
Attribute Symbol:
rts? ↦ *rts?*
msg ↦ *msg*
send_msg ↦ *send_msg*
Action Symbol:
Prod ↦ *Prod*
Send ↦ *Send*
Prod ↦ *Re_Prod*

Proof of well-definedness of S2S_R as specification homomorphism:

In order to prove that we have a morphism between the description of **Sender** to **Sender_Res** component, we have to prove that:

a The axioms of **Sender** are translated to theorems of **Sender_Res** AND

b The **Sender_Res** preserves the locality of **Sender**.

Proof a: The axioms S-1,S-2,S-3,S-4,S-5,S-6 are translated to S-R 1, S-R 2, S-R 4, S-R 6, S-R 7, and S-R 9 respectively.

Proof b: We will have to prove that, $(Prod \vee Re\_Prod \vee Send) \vee (\mathbb{X}\, rts? = rts? \wedge \mathbb{X}\, msg = msg \wedge \mathbb{X}\, send\_msg = send\_msg)$, i.e., the translation of the locality axiom for **Sender**.

1. $(Prod \vee Re\_Prod \vee Send) \vee (\mathbb{X}\, rts? = rts? \wedge \mathbb{X}\, msg = msg \wedge \mathbb{X}\, send\_msg = send\_msg \wedge \mathbb{X}\, lost\_msg = lost\_msg)$
   locality axiom for **Sender_Res**

2. *(Prod ∨ Re_Prod ∨ Send) ∨ (𝕏 rts? =rts? ∧ 𝕏 msg=msg ∧ 𝕏 Send_msg=send_msg)*
   1,PC

ST2S_R: SendTrans → Sender_Res

***Sort**:*
Bool ↦*Bool*,
Int ↦ *Int*
***Operators**:*
true ↦ *true*,
false ↦ *false*
***Attribute Symbol**:*
sync_simsg ↦ *send_msg*
***Action Symbol**:*
SyncSendAccept ↦ *Send*

Proof of well-definedness of ST2S_R as specification homomorphism:

In order to prove that we have a morphism between the description of **SendTrans** to **Sender_Res** component, we have to prove that:

a The axioms of **SendTrans** are translated to theorems of **Sender_Res** AND

b The **Sender_Res** preserves the locality of **SendTrans**.

Proof a: Since there is no axiom in **SendTrans**, axiom transformation holds vacuously.

Proof b: We will have to prove that, *(Send ∨ 𝕏 send_msg=send_msg)*, i.e., the translation of the locality axiom for **SendTrans**.

1. *(Prod ∨ Re_Prod ∨ Send) ∨ (𝕏 rts? =rts? ∧ 𝕏 msg=msg ∧ 𝕏 send_msg=send_msg ∧ 𝕏 lost_msg=lost_msg)*
   locality axiom for **Sender_Res**

2. *(Prod ∨ Re_Prod ∨ Send) ∨ (𝕏 send_msg=send_msg)*
   1,PC

3. *Send ∨ (𝕏 send_msg=send_msg)*
   S-R 4, S-R 5, 2, PC

T2TP: Trans → TransPlus

Sort:
Bool ↦*Bool*,
Int ↦ *Int*
Operators:
true ↦ *true*,
false ↦ *false*
Attribute Symbol:
rtt? ↦ *rtt?*,
in_msg ↦ *in_msg*,
acc_msg ↦ *acc_msg*,
tra_msg ↦ *tra_msg*
Action Symbol:
Accept ↦ *Accept*,
Transmit ↦ *Transmit*

Proof of well-definedness of T2TP as specification homomorphism:

In order to prove that we have a morphism between the description of **Trans** to **TransPlus** component, we have to prove that:

a The axioms of **Trans** are translated to theorems of **TransPlus** AND

b The **TransPlus** preserves the locality of **Trans**.

Proof a: The axioms T-1,T-2,T-3,T-4,T-5,T-6 are translated to TP-1, TP-3, TP-4,TP-5,TP-8 and TP-10 respectively.

Proof b: We will have to prove that, $(Accept \vee Transmit) \vee (\mathbb{X}\, rtt? = rtt? \wedge \mathbb{X}\, in\_msg = in\_msg \wedge \mathbb{X}\, acc\_msg = acc\_msg \wedge \mathbb{X}\, tra\_msg = tra\_msg)$, i.e., the translation of the locality axiom $(Accept \vee Transmit) \vee (\mathbb{X}\, in\_msg = in\_msg \wedge \mathbb{X}\, rtt? = rtt \wedge \mathbb{X}\, acc\_msg = acc\_msg \wedge \mathbb{X}\, tra\_msg = tra\_msg)$ for **Trans**.

1. $(Accept \vee Transmit\_M \vee Transmit) \vee (\mathbb{X}\, rtt? = rtt? \wedge \mathbb{X}\, in\_msg = in\_msg \wedge \mathbb{X}\, acc\_msg = acc\_msg \wedge \mathbb{X}\, tra\_msg\_m = tra\_msg\_m \wedge \mathbb{X}\, tra\_msg = tra\_msg)$,
   locality axiom for **TransPlus**
2. $(Accept \vee Transmit\_M \vee Transmit) \vee (\mathbb{X}\, rtt? = rtt? \wedge \mathbb{X}\, in\_msg = in\_msg \wedge \mathbb{X}\, acc\_msg = acc\_msg \wedge \mathbb{X}\, tra\_msg = tra\_msg)$,
   1,PC
3. $(Accept \vee Transmit) \vee (\mathbb{X}\, rtt? = rtt? \wedge \mathbb{X}\, in\_msg = in\_msg \wedge \mathbb{X}\, acc\_msg = acc\_msg \wedge \mathbb{X}\, tra\_msg = tra\_msg)$,
   TP-7,PC

TR2TP: TransRec → TransPlus

Sort:
Bool ↦*Bool*,
Int ↦ *Int*
Operators:
true ↦ *true*,
false ↦ *false*
Attribute Symbol:
sync_trmsg ↦ *tra_msg*
Action Symbol:
SyncTransRec ↦ *Transmit*

Proof of well-definedness of TR2TP as specification homomorphism:

In order to prove that we have a morphism between the description of **TransRec** to **TransPlus** (Transmitter Plus) components, we have to prove that:

a The axioms of **TransRec** are translated to theorems of **TransPlus** AND

b The **TransPlus** preserves the locality of **TransRec**.

Proof a: Since there is no axiom a holds vacuously.

Proof b: We will have to prove that, $Transmit \vee (\mathbb{X}\, tra\_msg = tra\_msg)$, i.e., the translation of the locality axiom $SyncTransRec \vee (\mathbb{X}\, sync\_trmsg = sync\_trmsg)$. The proof is given below:

1. $(Accept \vee Transmit\_M \vee Transmit) \vee (\mathbb{X}\, rtt? = rtt? \wedge \mathbb{X}\, in\_msg = in\_msg \wedge \mathbb{X}\, acc\_msg = acc\_msg \wedge \mathbb{X}\, tra\_msg\_m = tra\_msg\_m \wedge \mathbb{X}\, tra\_msg = tra\_msg)$,
   locality axiom for **TransPlus**
2. $(Accept \vee Transmit\_M \vee Transmit) \vee (\mathbb{X}\, tra\_msg = tra\_msg)$,
   1,PC
3. $(rtt? = true \wedge Transmit\_M) \rightarrow (\mathbb{X}\, tra\_msg\_m = acc\_msg \wedge \mathbb{X}\, in\_msg = in\_msg \wedge \mathbb{X}\, tra\_msg = tra\_msg)$,
   TP-4
4. $(rtt? = false \wedge Accept) \rightarrow (\mathbb{X}\, rtt? = true \wedge \mathbb{X}\, acc\_msg = in\_msg \wedge \mathbb{X}\, tra\_msg = tra\_msg)$,
   TP-7
5. $Transmit \vee (\mathbb{X}\, tra\_msg = tra\_msg)$
   3,4, PC

TM2TP: TransMontr → TransPlus

Sort:
Bool ↦ *Bool*,
Int ↦ *Int*
Operators:
true ↦ *true*,
false ↦ *false*,
Attribute Symbol:
sync_timsg ↦ *tra_msg_m*
Action Symbol:
SyncTrans(T)Accept(M) ↦ *Transmit_M*

Proof of well-definedness of TM2TP as specification homomorphism:

In order to prove that we have a morphism between the description of **TransMontr** to **TransPlus** component, we have to prove that:

a The axioms of **TransMontr** are translated to theorems of **TransPlus** AND

b The **TransPlus** preserves the locality of **TransMontr**.

Proof a: Since there is no axiom in **TransMontra** axiom transformation holds vacuously.

Proof b: We will have to prove that, $(Transmit \vee (\mathbb{X}\, tra\_msg\_m{=}tra\_msg\_m)$, i.e., the translation of the locality axiom $(SyncTrans(T)Accept(M) \vee (\mathbb{X}\, sync\_tra{-}in\_msg = sync\_tra{-}in\_msg)$ for **TransMontr**.

1. $(Accept \vee Transmit\_M \vee Transmit) \vee (\mathbb{X}\, rtt? = rtt? \wedge \mathbb{X}\, in\_msg{=}in\_msg \wedge \mathbb{X}\, acc\_msg = acc\_msg \wedge \mathbb{X}\, tra\_msg\_m{=}tra\_msg\_m \wedge \mathbb{X}\, tra\_msg{=}tra\_msg)$,
   locality axiom for **TransPlus**
2. $(Accept \vee Transmit\_M \vee Transmit) \vee (\mathbb{X}\, tra\_msg\_m{=}tra\_msg\_m)$,
   1,PC
3. $(Transmit\_M \vee \mathbb{X}\, tra\_msg\_m{=}tra\_msg\_m)$,
   TP-4, TP-5, TP-6, PC

TM2M: TransMontr → Monitor

Sort:
Bool ↦*Bool*,
Int ↦ *Int*
Operators:
true ↦ *true*,
false ↦ *false*
Attribute Symbol:
sync_timsg ↦ *in_msg*

Action Symbol:
SyncTrans(T)Accept(M) ↦ *Accept*

Proof of well-definedness of TM2M as specification homomorphism:

In order to prove that we have a morphism between the description of **TransMontr** to **Monitor** component, we have to prove that:

a The axioms of **TransMontr** are translated to theorems of **Monitor** AND

b The **Monitor** preserves the locality of **TransMontr**.

Proof a: Since there is no axiom in **TransMontra** axiom transformation holds vacuously.

Proof b: We will have to prove that, *(Accept ∨ (𝕏 in_msg=in_msg)*, i.e., the translation of the locality axiom *(SyncTrans(T)Accept(M)∨ (𝕏 sync_tra−in_msg =sync_tra −in_msg)* for **TransMontr**.

1. *(Accept ∨ Resend_Req) ∨ (𝕏 rtm? =rtm? ∧ 𝕏 in_msg=in_msg ∧ 𝕏 rst_msg=rst_msg ∧ 𝕏 exp_msg=exp_msg)*,
   locality axiom for **Monitor**
2. *(Accept ∨ Resend_Req) ∨ (𝕏 in_msg=in_msg)*,
   1,PC
3. *(Accept ∨ 𝕏 in_msg=in_msg)*,
   2, M-6, PC

MS_R2M: MontrSend_Res → Monitor

Sort:
Bool ↦ *Bool*,
Int ↦ *Int*
Operators:
true ↦ *true*,
false ↦ *false*,
Attribute Symbol:
sync_rlmsg ↦ *rst_msg*

Action Symbol:
SyncResendProd ↦ *Resend_Req*

Proof of well-definedness of MS_R2M as specification homomorphism:

In order to prove that we have a morphism between the description of **MontrSend_Res** to **Monitor** component, we have to prove that:

a The axioms of **MontrSend_Res** are translated to theorems of **Monitor** AND

b The **Monitor** preserves the locality of **MontrSend_Res**.

Proof a: Since there is no axiom in **MontrSend_Res** axiom transformation holds vacuously.

Proof b: We will have to prove that, *(Resend_Req* ∨ *(*$\mathbb{X}$ *rst_msg=rst_msg)*, i.e., the translation of the locality axiom *SyncResendProd* ∨ *(*$\mathbb{X}$ *sync_rst−lost_msg =sync_rst −lost_msg)* for **MontrSend_Res**.

1. *(Accept* ∨ *Resend_Req)* ∨ *(*$\mathbb{X}$ *rtm? =rtm?* ∧ $\mathbb{X}$ *in_msg=in_msg* ∧ $\mathbb{X}$ *rst_msg=rst_msg* ∧ $\mathbb{X}$ *exp_msg=exp_msg)*,
   locality axiom for **Monitor**
2. *(Accept* ∨ *Resend_Req)* ∨ *(*$\mathbb{X}$ *rst_msg=rst_msg)*,
   1,PC
3. *(Resend_Req* ∨ $\mathbb{X}$ *rst_msg=rst_msg)*,
   2, M-4, M-5, PC

MS_R2S_R: MontrSend_Res → Sender_Res

Sort:
Bool ↦*Bool*,
Int ↦ *Int*
Operators:
true ↦ *true*,
false ↦ *false*
Attribute Symbol:
sync_rlmsg ↦ *lost_msg*

Action Symbol:
SyncResendProd ↦ *Re_Prod*

Proof of well-definedness of MS_R2S_R as specification homomorphism:

In order to prove that we have a morphism between the description of **MontrSend_Res** to **Sender_Res** component, we have to prove that:

a The axioms of **MontrSend_Res** are translated to theorems of **Sender_Res** AND

b The **Sender_Res** preserves the locality of **MontrSend_Res**.

Proof a: Since there is no axiom in **MontrSend_Res** axiom transformation holds vacuously.

Proof b: We will have to prove that, *Re_Prod* ∨ *(*$\mathbb{X}$ *lost_msg=lost_msg)*, i.e., the translation of the locality axiom *SyncResendProd* ∨ *(*$\mathbb{X}$ *sync_rst−lost_msg =sync_rst −lost_msg)* for **MontrSend_Res**.

1. *(Prod* ∨ *Re_Prod* ∨ *Send)* ∨ *(*$\mathbb{X}$ *rts? =rts?* ∧ $\mathbb{X}$ *msg=msg* ∧ $\mathbb{X}$ *Send_msg=Send_msg* ∧ $\mathbb{X}$ *lost_msg=lost_msg)*
   locality axiom for **Sender_Res**

2. *(Prod* ∨ *Re_Prod* ∨ *Send)* ∨ *(*$\mathbb{X}$ *lost_msg=lost_msg)*
   1,PC

3. *(Re_Prod* ∨ $\mathbb{X}$ *lost_msg=lost_msg)*,
   2, S-R 4, S-R 6, PC

B.3.5 Reliable System Specification

UnsecRelSyst:

The signature of secured-unreliable system **UnsecRelSyst** would be as follows:

Sort:

Bool, *Int*

Operators:

true, *false* : *Bool*
$+$: *Int*, *Int* $\rightarrow$ *Int*
notZero : *Int* $\rightarrow$ *Bool*
isEfree : *Int* $\rightarrow$ *Bool*
isExpMessage: *Int*, *Int* $\rightarrow$ *Bool*

Attribute Symbol:

rts? : *Bool*
msg : *Int*
sync_simsg : *Int*
sync_rlmsg : *Int*
rtt? : *Bool*
acc_msg : *Int*
sync_trmsg : *Int*
sync_timsg : *Int*
rtr? : *Bool*
final_msg : *Int*
rtm? : *Bool*
exp_msg : *Int*

Action Symbol:

SyncResendProd
SyncSendAccept
SyncTransRec
SyncTrans(T)Accept(M)
Re−state

We can describe the unsecured-reliable system **UnsecRelSyst** by the following axioms:

M- 1: ***Beg*** $\rightarrow$ *rtm? = true*

M- 2: ***Beg*** $\rightarrow$ *sync_rlmsg = 0*

M- 3: ***Beg*** $\rightarrow$ *exp_msg= 0*

M- 4: $(rtm? = true \wedge isExpMessage(exp\_msg,sync\_timsg) = true \wedge SyncTrans(T)Accept(M)) \rightarrow (\mathbb{X}\, exp\_msg = exp\_msg + 1)$

M- 5: $(rtm? = true \wedge isExpMessage(exp\_msg,sync\_timsg) = false \wedge SyncTrans(T)Accept(M)) \rightarrow (\mathbb{X}\, exp\_msg = exp\_msg \wedge rtm? = false)$

M- 6: $(rtm? = false \wedge SyncResendProd) \rightarrow (\mathbb{X}\, sync\_rlmsg = exp\_msg \wedge rtm? = true)$

M- 7: $(rtm? = true\ ) \rightarrow \mathbb{F}\, SyncTrans(T)Accept(M)$

M- 8: $(rtm? = false\ ) \rightarrow \mathbb{F}\, SyncResendProd$

R- 1: $\boldsymbol{Beg} \rightarrow rtr? = true$

R- 2: $\boldsymbol{Beg} \rightarrow final\_msg = 0$

R- 3: $(rtr? = true \wedge SyncTransRec) \rightarrow (\mathbb{X}\, rtr? = false \wedge \mathbb{X}\, final\_msg = sync\_trmsg)$

R- 4: $(rtr? = false \wedge Re{-}state) \rightarrow (\mathbb{X}\, rtr? = true \wedge \mathbb{X}\, sync\_trmsg = sync\_trmsg)$

R- 5: $(rtr? = true) \rightarrow \mathbb{F}\, SyncTransRec$

R- 6: $(rtr? = false) \rightarrow \mathbb{F}\, Re{-}state$

TP- 1: $\boldsymbol{Beg} \rightarrow rtt? = false$

TP- 2: $\boldsymbol{Beg} \rightarrow tra\_msg\_m = 0$

TP- 3: $\boldsymbol{Beg} \rightarrow tra\_msg = 0$

TP- 4: $(rtt? = false \wedge SyncSendAccept) \rightarrow (\mathbb{X}\, rtt? = true \wedge \mathbb{X}\, acc\_msg = sync\_simsg \wedge \mathbb{X}\, sync\_trmsg = sync\_trmsg)$

TP- 5: $(rtt? = true \wedge isEfree\ (acc\_msg) = true \wedge SyncTransRec) \rightarrow (\mathbb{X}\, sync\_trmsg = acc\_msg \wedge \mathbb{X}\, rtt? = false \wedge \mathbb{X}\, in\_msg = in\_msg)$

TP- 6: $(rtt? = true \wedge isEfree\ (acc\_msg) = false) \rightarrow (\neg\, \mathbb{F}\, SyncTransRec)$

TP- 7: $(rtt? = true \wedge SyncTrans(T)Accept(M)) \rightarrow (\mathbb{X}\, sync\_timsg = acc\_msg \wedge \mathbb{X}\, in\_msg = in\_msg)$

TP- 8: $(rtt? = false) \rightarrow \mathbb{F}\, SyncSendAccept$

TP- 9: $(rtt? = true) \rightarrow \mathbb{F}\, SyncTrans(T)Accept(M)$

TP- 10: $(rtt? = true \wedge isEfree\ (acc\_msg) = true) \rightarrow \mathbb{F}\, SyncTransRec$

S-R 1: $\boldsymbol{Beg} \rightarrow rts? = false$

S-R 2: $\boldsymbol{Beg} \rightarrow msg = 0$

S-R 3: $\boldsymbol{Beg} \rightarrow sync\_rlmsg = 0$

S-R 4: $(rts? = false \wedge notZero(sync\_rlmsg)=false \wedge SyncResendProd\ ) \rightarrow \mathbb{X}\, rts? = true \wedge \mathbb{X}\, msg = msg + 1 \wedge \mathbb{X}\, sync\_simsg=sync\_simsg \wedge \mathbb{X}\, sync\_rlmsg=sync\_rlmsg)$

S-R 5: $(rts? = false \wedge notZero(sync\_rlmsg)=true \wedge SyncResendProd) \rightarrow (\mathbb{X}\, rts? = true \wedge \mathbb{X}\, msg = sync\_rlmsg \wedge \mathbb{X}\, sync\_simsg=sync\_simsg \wedge \mathbb{X}\, sync\_rlmsg = 0)$

S-R 6: $(rts? = true \wedge SyncSendAccept) \rightarrow (\mathbb{X}\, sync\_simsg= msg \wedge \mathbb{X}\, rts? = false \wedge \mathbb{X}\, msg=msg \wedge \mathbb{X}\, sync\_rlmsg=sync\_rlmsg)$

S-R 7: $(rts? = false) \rightarrow \mathbb{F}\, SyncResendProd$

S-R 8: $(rts? = true) \rightarrow \mathbb{F}\, SyncSendAccept$

B.4 Graph Transformation Rules

For any giving arbitrary architecture where unsecured communication exist, if we want to introduce security on it by performing graph transformation, the transformation rule we will apply are as follows:

Security Introduction:

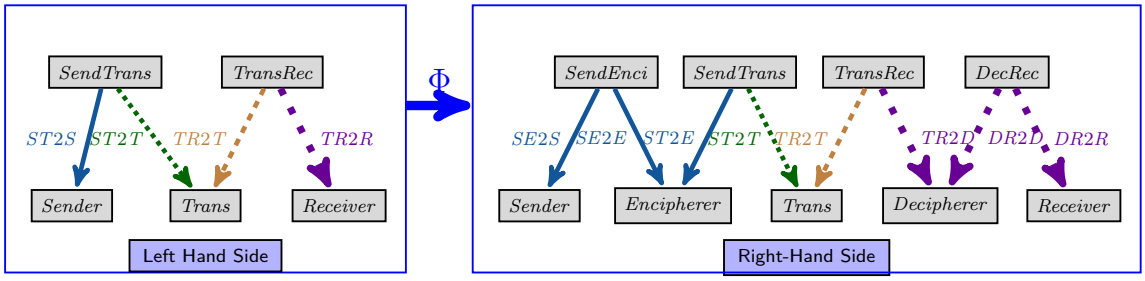


Figure B.4.1: Rule: Security Introduction

For any giving arbitrary architecture where unreliable communication exist, if we want to introduce reliability on it by performing graph transformation, the transformation rule we will apply is depicted in the follows Figure B.4.2. The view of the right architecture in reliability introduction rule is different from the reliable communication architecture (developers view) depicted in Figure B.3.1.

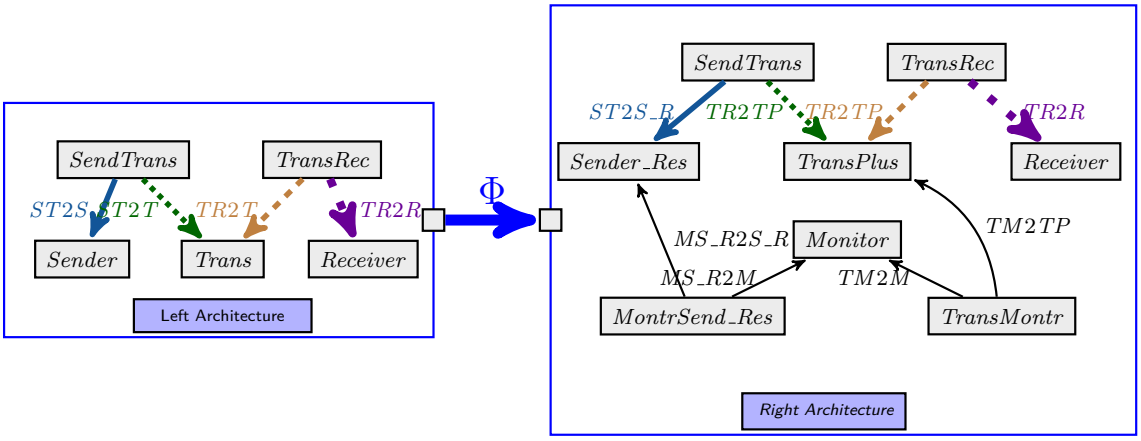


Figure B.4.2: Rule: Reliability Introduction

Reliability Introduction:

B.5 Architecture With Aspect

In the following graph, all the nodes are the component specification of a given architecture and the edges are the total specification morphisms (preserving locality) between components. The morphisms between graphs are *Zpath* graph homomorphisms.

Suppose we have an architecture where two cars communicate with each other. In order to make it simple, lets say a lead vehicle measures the speed of a following vehicle as well as the distance between two and sends some signal to reduce speed for avoiding collision through sender to transmitter. Transmitter transmit it to the target vehicle. The following vehicle receives the signal and send instruction to break sensor to break. The architecture of this system can be represented as follows:

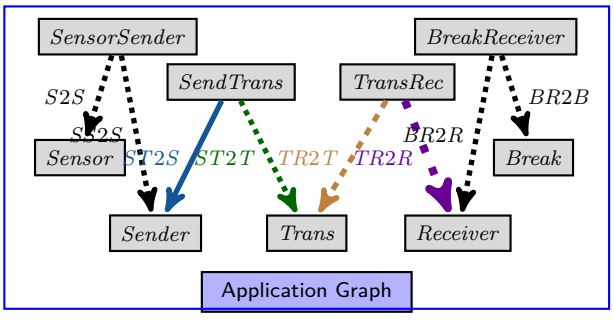


Figure B.5.1: A Bigger Architecture

B.5.1 Secured Sender Receiver Communication

For a given transformation rule (*Security Introduction*) and an application graph,an architecture where unsecured message sending and receiving communication exist, we can make this communication secure by performing graph transformation. The application of the transformation will give us the following pushout diagram:

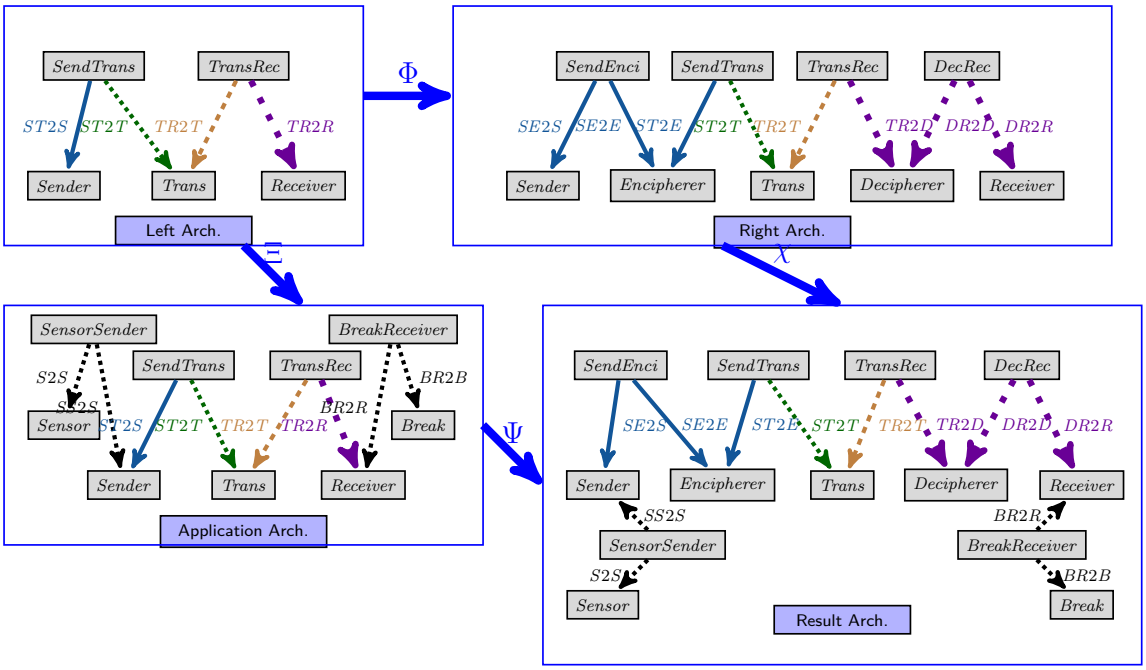


Figure B.5.2: Secured Communication Architecture

B.5.2 Reliable Sender Receiver Communication

For a given transformation rule (*Reliability Introduction*) and an application graph where unreliable communication exist, In order to introduce reliability to the application graph (bigger architecture) if we apply transformation,it will give us the following pushout diagram and resultant graph (pushout object):

B.5.3 Introduce Security on Unsecured-Reliable Communication

Consider we have an Unsecured-Reliable architecture (application graph) and a transformation rule *(Security Introduction)*. If we apply the graph transformation technique to introduce security to our Unsecured-Reliable architecture, the transformation will produce the following result graph.

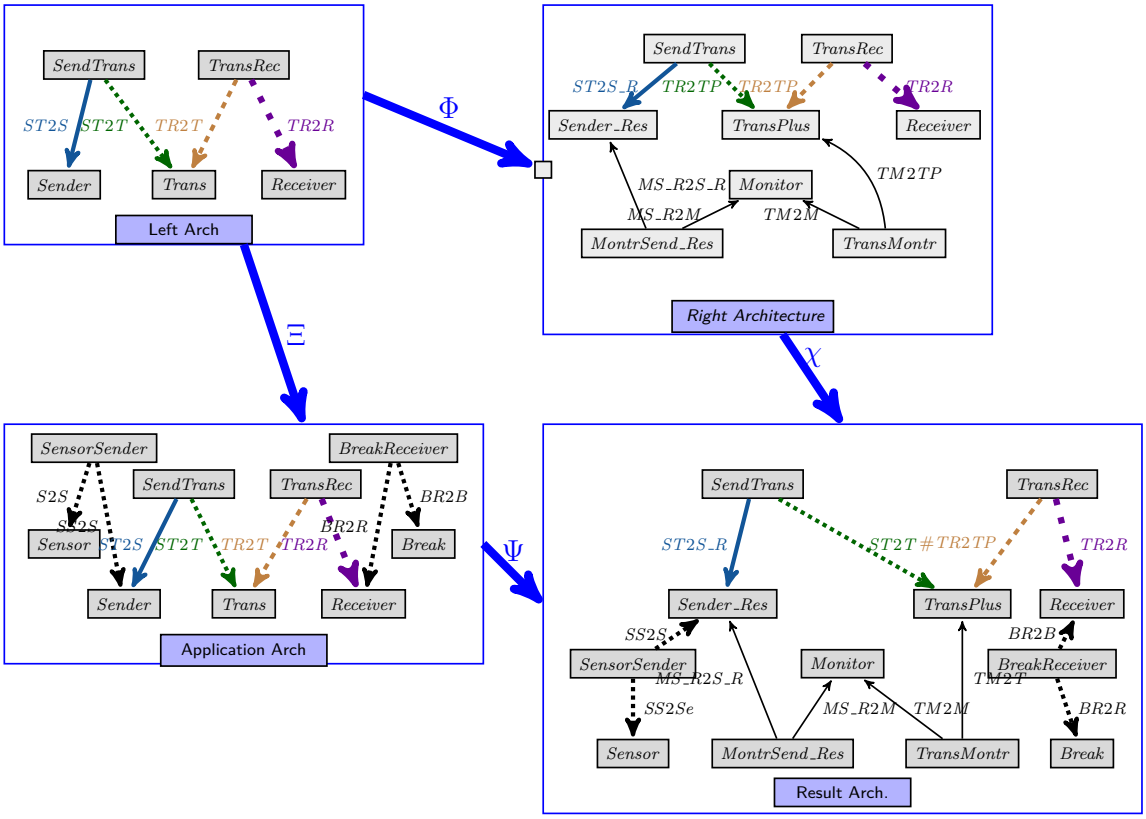


Figure B.5.3: Aspect Intro: Reliable Communication Architecture

B.5.4 Specification of the New Signature Morphism and well-definedness

SE2S_R: SendEnci → Sender_Res

Sort:

$Bool \mapsto Bool$,

$Int \mapsto Int$

Operators:

$true \mapsto true$,

$false \mapsto false$,

Attribute Symbol:

$sync\_csmsg \mapsto send\_msg$

Action Symbol:

$SyncSendConcede \mapsto Send$

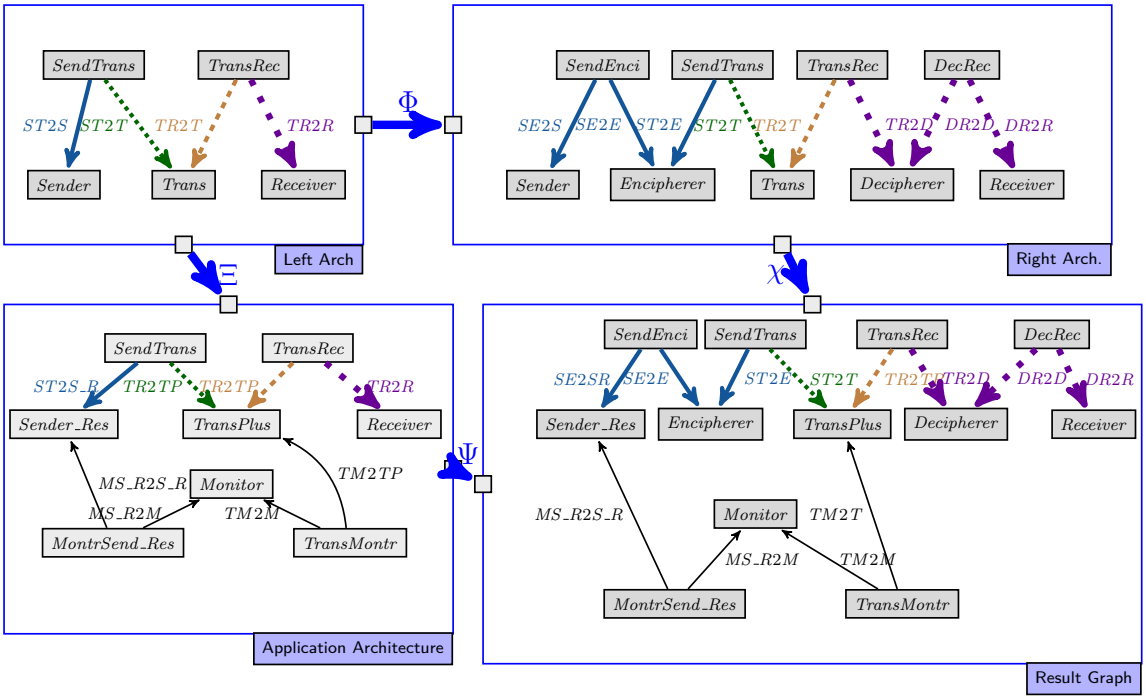


Figure B.5.4: Reliable-Secured Communication

Proof of well-definedness of SE2S_R as specification homomorphism:

In order to prove that we have a morphism between the description of **SendEnci** to **Sender_Res** component, we have to prove that:

a The axioms of **SendEnci** are translated to theorems of **Sender_Res** AND

b The **Sender_Res** preserves the locality of **SendEnci**.

Proof a: Since there is no axiom a holds vacuously.

Proof b: We will have to prove that, $Send \vee (\mathbb{X}\, send\_msg = send\_msg)$, i.e., the translation of the locality axiom $SyncSendConcede \vee (\mathbb{X}\, sync\_scmsg = sync\_scmsg)$. The proof is given below:

1. $(Prod \vee Re\_Prod \vee Send) \vee (\mathbb{X}\, rts? = rts? \wedge \mathbb{X}\, msg = msg \wedge \mathbb{X}\, Send\_msg = Send\_msg \wedge \mathbb{X}\, lost\_msg = lost\_msg)$
   locality axiom for **Sender_Res**

2. $(Prod \vee Re\_Prod \vee Send) \vee (\mathbb{X}\, send\_msg = send\_msg)$
   1,PC

3. *Send* $\vee$ *(*$\mathbb{X}$ *send_msg =send_msg)*
   S-R 4, S-R 5, 2 PC

B.5.5 Introduce Reliability on Unreliable-Secured Communication

Now consider that, we have an Unreliable-Secured architecture (application graph) and a transformation rule *(Reliability Introduction)*. We want to introduce Reliability to our complex architecture by performing graph transformation. If we apply our technique the transformation will produce the following result graph.

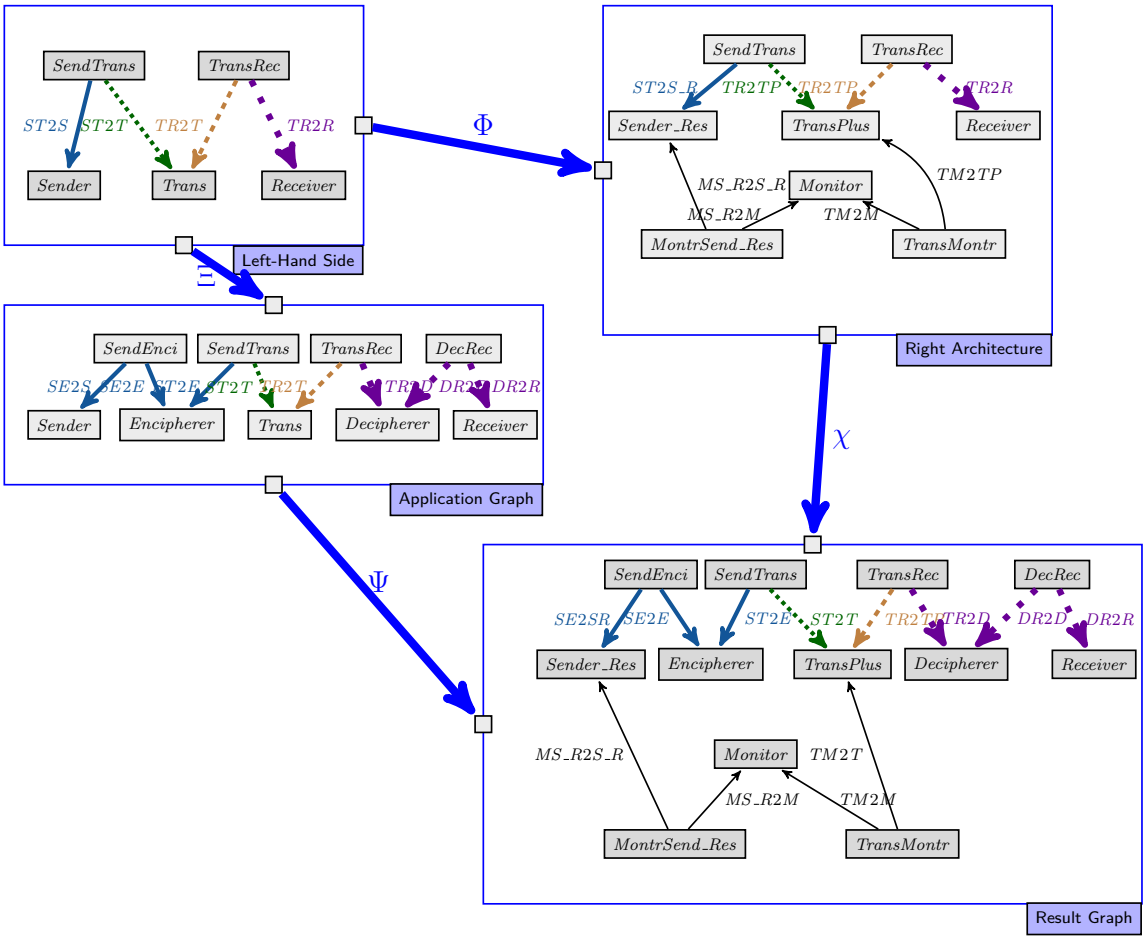


Figure B.5.5: Secured-Reliable Communication