Towards Architecture-Centric Evolution of Connectors

Selva Samuel

CMU-S3D-24-108

October 2024

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Prof. Jonathan Aldrich, Chair
Prof. David Garlan
Prof. Eunsuk Kang
Prof. Nenad Medvidović (USC)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.

This work was sponsored by the National Science Foundation (NSF) under grant no. CCF1901033, the Defense Advanced Research Projects Agency (DARPA) under agreement no. FA8750-16-2-0042, the National Security Agency Department of Defense awards H9823014C0140 and H9823018D0009, the Algorand Foundation, and Ripple. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the NSF, DARPA, NSA, Algorand Foundation, Ripple, the U.S. government or any other entity.

Keywords: Software architecture, connectors, software evolution, connector evolution, architecture-centric development

Abstract

As a software system evolves, the interactions between different parts of the system, which represent connectors in the runtime architecture, might need to be changed to meet changing requirements. In current practice, such changes need to be made directly in code. However, changing connectors directly in code can be tedious and error-prone because it involves modifying many lines of code spread across multiple files. These challenges exist because current programming languages lack an abstraction mechanism for modularizing connectors. In this thesis, we lay out an architecture-centric development approach that mitigates these challenges by introducing an abstraction mechanism that decouples connectors from functional code, making it easy to change or evolve connectors. This approach additionally involves integrating the architecture specification of a system with its implementation. We have implemented this approach in a programming language that we have developed called Wyvern.

When connectors need to be changed, architects and developers might come up with a set of candidate connectors that could satisfy the new requirements. They might be interested in knowing which of the candidate connectors are semantically compatible with the connector to be changed. Checking semantic compatibility involves ensuring that one or more behaviors of interest in the original connector are preserved in the new connector as well. Compatibility could be assessed along several dimensions. In this thesis, we focus on *data relay compatibility*. Data relay compatibility means that every input data accepted by the original connector is also accepted by the new connector and is sent to at least the same outputs as the original connector. To check data relay compatibility, we specify the data transmission behavior of connectors using constraint automata. Our compatibility checking algorithm is based on symbolic execution of constraint automata. We have extended the connector abstraction mechanism in Wyvern to support constraint automata specifications. We have also implemented a tool that would enable architects and developers to perform compatibility analyses based on the constraint automata specifications.

We evaluate the effectiveness of our architecture-centric development approach in making the task of connector evolution in two case studies. We demonstrate through the two case studies that our approach makes the connector evolution task easy in situations where component interfaces remain unchanged. We also demonstrate the generality of the connector abstraction mechanism we have developed by implementing several common types of connectors using the mechanism. Additionally, we show how the data transmission semantics of several connectors may be encoded using constraint automata and demonstrate the usefulness of our connector compatibility analysis approach in preventing errors during connector evolution by catching them early in the evolution process.

Acknowledgments

This long journey has finally come to an end and I'm grateful to the many people who helped me finish my PhD.

First and foremost, I would like to thank my advisor, Prof. Jonathan Aldrich, for his incredible support, guidance and patience throughout my time as a PhD student. He gave me the freedom to pursue my ideas and develop as an independent researcher.

I thank the other members of my thesis committee, Prof. David Garlan, Prof. Eunsuk Kang and Prof. Nenad Medvidović, for their input and invaluable feedback that have made my work better. They have been very understanding of my situation in the final months of finishing this dissertation and have been very accommodating about the short timeline within which I needed to defend and graduate.

I'm thankful to my incredible friend, Mandy, who provided me financial support in the final months of completing this dissertation. I will forever be grateful to her.

Finally, I would like to thank my beloved partner, Catherine, who has been very understanding when I've had to spend long hours working on this dissertation. She has provided me encouragement and stood by me during troublesome times. She continued to believe in me when I couldn't. I wouldn't have finished my PhD if not for her support.

Contents

1 Introduction **1**
- 1.1 Thesis Statement . . . 3
- 1.2 Thesis Contributions . . . 4
- 1.3 Dissertation Outline . . . 5

2 Related Work **7**
- 2.1 Software Architecture . . . 7
  - 2.1.1 Connectors . . . 7
- 2.2 Taxonomy . . . 8
- 2.3 Connector Code Generation . . . 8
  - 2.3.1 Connector Implementations . . . 8
- 2.4 Connector Evolution . . . 9
- 2.5 Connector Compatibility . . . 10

3 Connector Evolution Approach **11**
- 3.1 Motivating Example . . . 11
- 3.2 Goals . . . 13
- 3.3 Architecture-Centric Development Approach . . . 14
  - 3.3.1 Architecture Description . . . 16
  - 3.3.2 Integration of Architecture Description with Code . . . 17
  - 3.3.3 Connector Modification . . . 21

4 Connector Compatibility Analysis **23**
- 4.1 Connector Compatibility . . . 23
- 4.2 Connector Compatibility Analysis Approach . . . 25
  - 4.2.1 Constraint Automata . . . 25
  - 4.2.2 Symbolic Execution of Constraint Automata . . . 28
  - 4.2.3 Checking Data Relay Compatibility . . . 32
  - 4.2.4 Composition of Constraint Automata . . . 33
- 4.3 Implementation . . . 37

5 Evaluation **39**
- 5.1 Claim 1: Ease of Connector Evolution Via Architecture-Code Integration . . . 40
  - 5.1.1 Case Study 1: ROS 1 to ROS 2 Migration . . . 40

5.1.2 Case Study 2: SQL to NoSQL Migration 51
5.1.3 Discussion 65
5.2 Claim 2: Generality of the Connector Abstraction Mechanism and the Connector Evolution Approach 67
5.2.1 Generality of the Connector Abstraction Mechanism 67
5.2.2 Generality of the Connector Evolution Approach 71
5.3 Claim 3: Usefulness of Connector Compatibility Analysis in Prevention of Errors 74
5.3.1 Call Synchronicity 75
5.3.2 Delivery Policy 76
5.3.3 Message Delivery Order 77
5.3.4 Limitations of the Connector Compatibility Analysis Approach 78
5.4 Summary 78

6 Conclusion and Future Work **79**

Bibliography **81**

List of Figures

1.1 An example client-server system 1

3.1 Migration of a producer-consumer system from RabbitMQ to Apache Kafka . . . 12
3.2 Changing connectors in our architecture-centric development approach 15
3.3 Architecture description of the RabbitMQ-based producer-consumer system . . . 17
3.4 Module definitions for the `Producer` and `Consumer` component types 18
3.5 Type definition corresponding to the `RabbitMQQueue` connector type 19
3.6 Module generated for the `requires` port with `StringDataIface` interface connected by a `RabbitMQQueue` connector 20
3.7 Architecture description and connector abstraction for Apache Kafka-based producer-consumer system 21

4.1 An example connector and its constraint automaton 29
4.2 A simple system with two pipe connectors that need to be replaced 34
4.3 Constraint automata for the two pipe connectors in the system whose architecture is shown in figure 4.2 34
4.4 Composite constraint automaton for the two constraint automata shown in figure 4.3 36
4.5 Constraint automaton specification using the DSL in WYVERN 38
4.6 Constraint automaton specification in connector type metadata 38

5.1 Architecture of the TurtleBot3 teleoperation system 42
5.2 Data model showing the cardinality relationships among the tables in the database used in the SCARF application 52
5.3 State machine for the operation of a circuit breaker 70
5.4 Constraint automata for (b) asynchronous and (c) synchronous RPC connectors that may be used in a client/server system 75
5.5 Constraint automata for event connectors with (b) guaranteed delivery policy and (c) best effort delivery policy that may be used in a producer/consumer system . . 76
5.6 Constraint automata for buffered stream connectors with (b) oldest first delivery order and (c) most recent first delivery order that may be used in a producer/consumer system 77

List of Tables

5.1 ROS communication primitives included in the systems used in the ROS 1 to ROS 2 migration case study 44
5.2 Changes made to migrate connectors in the TurtleBot3 Teleoperation system from ROS 1 to ROS 2 45
5.3 Changes made to migrate connectors in the PX4 Control system from ROS 1 to ROS 2 47
5.4 Changes made to migrate connectors in the BB8 Square Motion system from ROS 1 to ROS 2 47
5.5 Changes made to migrate connectors in the Parrot Drone Square Motion system from ROS 1 to ROS 2 48
5.6 Changes made to migrate connectors in the Robotiq Gripper Control system from ROS 1 to ROS 2 48
5.7 Comparison between the changes made to the WYVERN implementation of SCARF with the abstract port interfaces and the changes made to the original implementation to migrate them to the MongoDB database with an unchanged schema 57
5.8 Comparison between the changes made to the WYVERN implementation of SCARF with the abstract port interfaces and the changes made to the original implementation to migrate them to the MongoDB database with a changed schema 58
5.9 Comparison between the changes made to the WYVERN implementation of SCARF with the SQL-specific port interfaces and the changes made to the original implementation to migrate them to the MongoDB database with an unchanged schema 61
5.10 Comparison between the changes made to the WYVERN implementation of SCARF with the SQL-specific port interfaces and the changes made to the original implementation to migrate them to the MongoDB database with a changed schema 63
5.11 Connectors implemented using the connector abstraction mechanism in WYVERN 71
5.12 Connector evolution scenarios implemented using our connector evolution approach 74

Chapter 1

Introduction

The runtime architecture of a system, consisting of *components* and *connectors*, is an important aspect of software design [46, 97]. Components are the computational elements and data stores that are present in the system. Connectors describe the mechanisms by which the components in the system interact. For example, in a web-based client-server application, the clients and server make up the components of the system, while the HTTP(S) protocol used to communicate between them is the connector (see Figure 1.1).

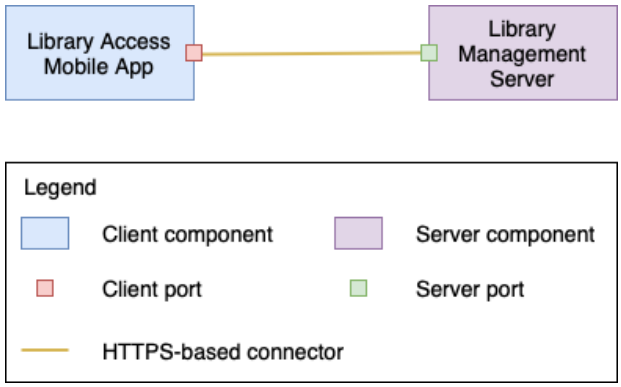


Figure 1.1: An example client-server system

The runtime architecture of a software system has a considerable influence on how well the system satisfies its requirements, in particular its quality attribute requirements. As a result, the runtime architecture specification enables reasoning about the properties of the system such as latency [56], throughput [57], scalability [43, 63], reliability [49, 60], etc. Moreover, analysis of architecture models allows errors to be detected early in the development process.

It is well known that a software system must constantly be modified to meet changing requirements [64]. As Lehman has noted in his laws of software evolution, a software system[1] must be continually adapted if it is to continue to remain useful [48, 51, 62]. Evolution of software systems is driven by various factors. A software system may be modified to add new features or to modify or even remove existing features. It may be changed to fix defects or to better meet

[1] Strictly, an E-type system [51]. We consider only such systems in this thesis.

quality attribute requirements such as security or performance. Changes might be carried out to improve the user experience. A software system might be changed to adopt new technologies, platforms, standards and protocols. Changes might be undertaken to make software systems work together even if they weren't initially intended to do so [24].

As a software system evolves, its runtime architecture may need to be changed. Architectural changes are *structural changes* in that they modify the structure of the software system [36]. As such, these changes might involve addition, removal and replacement of components or connectors [71, 84, 104, 107].

In recent years, connector evolution has become a key concern in the architectural evolution of software systems. For example, the need to process ever-increasing volumes of data with good performance requires migrating applications from relational databases to NoSQL databases. This, in turn, necessitates changing the application code that interacts with the database. The connectors in a software system might need to be changed to improve the quality attributes of the system such as performance, security, etc. [80, 104].

The evolution of connectors, however, presents several challenges. First, the implementation of the connector's functionality—initialization, communication of data, and cleanup—is spread across the codebase. Changing the connector thus involves modifying multiple lines of code spread across multiple files, a tedious and error-prone process. For example, ROS [88] has emerged as a popular middleware for implementing robotics software systems in recent years. ROS supports communication between components (called nodes) using publish/subscribe (called topics) and RPC request/reply (called services) connector mechanisms. ROS comes in two flavors - ROS 1 and ROS 2. In contrast to ROS 1, the communication between nodes in ROS 2 can be made reliable. Also, real-time guarantees can be provided in ROS 2. If these features are required for connectors in existing ROS 1 systems, then these systems must be updated to use ROS 2 connectors instead. A typical ROS 1 system consists of several source files that implement the nodes in the system. In the implementation of each node, there would be several lines corresponding to the initialization, use and cleanup of one or more topics and services. To migrate the system to use ROS 2 connectors, all of these lines littered across the implementation of each node must be changed which can be tedious and error-prone.

Second, connectors may depend on middleware, so changing the former often entails changing the latter as well. Migrating an application to use a different middleware is, however, a challenging task because of the tight coupling between the application code and the middleware. In the ROS example described above, the implementation of the connectors depend upon the underlying ROS middleware's libraries. Thus, updating a system to use ROS 2 connectors instead of ROS 1 connectors results in a change of the underlying middleware from ROS 1 to ROS 2. Changing the middleware from ROS 1 to ROS 2 is difficult because of the tight coupling between the connector implementation and the ROS middleware.

Additionally, for many systems, compatibility with previous versions is important [68]. In the context of connector evolution, compatibility means that certain behaviors of interest in the original connector are preserved by the new connector as well. However, currently reasoning about connector compatibility is a manual process.

In this thesis, I propose an architecture-centric approach to software development for addressing these challenges. This approach integrates the runtime architecture description of a system with its implementation. This allows component interaction code to be generated from the

connector specification in the architecture description. Connectors can then be easily changed simply by changing the architecture specification. Moreover, middleware dependencies are encapsulated into the connector as well, making it easy to change connectors even when their implementation is tied to a particular middleware. The integration of the architecture specification with the implementation also enables automated checking of compatibility between candidate connectors during evolution. This analysis can be used to plan how the changes required for connector evolution can be carried out.

1.1 Thesis Statement

This thesis will demonstrate that

> *We can make the evolution of connectors in a software system easier by integrating the architecture description of the system with its implementation when building the system. The integration of the architecture description of a system with its implementation can be achieved in a general way so that a wide variety of connectors as well as connector evolution scenarios may be supported. Furthermore, this integration facilitates analysis of the semantic compatibility of different connectors which can aid in the evolution of connectors by enabling architects to assess whether a candidate replacement connector preserves a behavior of interest and thus prevent unintended introduction of errors.*

The thesis statement is composed of three separate claims.

First, "we can make the evolution of connectors in a software system easier by integrating the architecture description of the system with its implementation when building the system." By evolution of connectors, we mean replacing one connector with another in order to improve one or more properties of the system. Our goal is to make this task easier compared to the current practice of changing connectors by directly changing code. We do this by localizing the changes to be made to a few locations in the codebase in contrast to the widespread changes to the codebase that need to be performed currently. This can be achieved by integrating the architecture description of the system with its implementation when building the system. This integration involves defining interfaces for the components. Components must then be implemented to interact exclusively via these interfaces. To do this, there also needs to be a mechanism to prevent component interaction without the use of their interfaces. Additionally, a mechanism for localizing connector implementation is needed so that the connector implementation is independent of the component interfaces.

Second, "the integration of the architecture description of a system with its implementation can be achieved in a general way so that a wide variety of connectors as well as connector evolution scenarios may be supported." In this thesis, we define a general abstraction mechanism that is expressive enough to implement the connectors in use in real-world systems today. Mehta, Medvidovic and Phadke [75] have created a taxonomy of software connectors based on the connectors that are used in real-world systems. We will show the generality of our abstraction mechanism by implementing representative example connectors from this taxonomy. We also identify categories of connector evolution scenarios. We will show how our approach supports each of these categories.

Third, "this integration facilitates analysis of the semantic compatibility of different connectors which can aid in the evolution of connectors by enabling architects to assess whether a candidate replacement connector preserves a behavior of interest and thus prevent an unintended introduction of errors." The behavioral semantics of connectors can be encoded in the abstraction mechanism that we develop to localize the implementation of connectors to enable connector evolution as laid out in the first claim above. This encoding of connector semantics enables checking if two connectors are compatible with each other. The results of this analysis can be used by architects to determine whether the candidate replacement connector they're considering is suitable for their purposes.

1.2 Thesis Contributions

The contributions of this thesis are as follows:

- **Architecture-Centric Connector Evolution Approach:** The primary contribution of this thesis is the development of an architecture-centric development approach that makes the task of connector evolution easier. Our approach involves the integration of the architecture description of a system with its implementation when building the system. We also provide an explicit abstraction mechanism for connectors. Components interact only through interfaces and the code implementing a connector is generated using its abstraction. As a result, the implementation of a connector is not spread across the codebase as happens in the current state of practice. Because of this, changing a connector can be achieved by changing just the architecture description if the component interfaces aren't changed.
- **Mechanism for Preventing Bypass of Connector Abstractions:** We have developed a mechanism for preventing developers from bypassing connector abstractions that's based on capabilities. This prevents them from directly using libraries for connector implementation. Since developers are forced to use connector abstractions, connector implementation code is prevented from being spread all over the codebase as happens when implementing a system using current programming languages.
- **Support for Specification of Data Transfer Semantics of Connectors:** To support architects in evaluating whether the replacement connectors they're considering preserve the data transfer behavior of a connector they want to change, we provide a mechanism for specifying the data transfer semantics of a connector in its abstraction. We use constraint automata for the specification of the data transfer semantics of a connector.
- **Connector Compatibility Analysis Algorithm:** We have provided an algorithm for checking whether a replacement connector is compatible with a connector that needs to be replaced with respect to the data transfer semantics. Our algorithm is based on the symbolic execution of the constraint automata corresponding to the connectors.
- **Empirical Validation:** We have evaluated the effectiveness of our connector evolution approach in making the task of connector evolution easy in two case studies. The results from the two case studies show that our approach makes the task of connector evolution much easier in cases where the component interfaces need not be changed. In these cases, the changes to be made are completely localized to the architecture description. We have

shown the expressiveness of our connector abstraction mechanism by implementing a wide range of connectors using the mechanism. We have also shown the generality of our connector evolution approach by using it to change connectors in a wide range of scenarios. Finally, we have demonstrated the usefulness of our connector compatibility analysis approach in preventing errors during connector evolution by presenting examples of the class of incompatibilities that can be detected using our approach.

1.3 Dissertation Outline

The rest of the dissertation is structured as follows. Chapter 2 discusses the related work that this thesis builds on. Chapter 3 presents our architecture-centric evolution approach. Chapter 4 presents our connector compatibility analysis approach which is based on symbolic execution of constraint automata that specify the data transfer semantics of connectors. Chapter 5 discusses how the claims of our thesis statement are validated. Chapter 6 concludes with a discussion of future work.

Chapter 2

Related Work

2.1 Software Architecture

The research in this thesis builds on past work in several areas. It is founded on software architecture, particularly on the idea of software connectors as first-class citizens.

Software architecture deals with the description of a software system abstractly in terms of *components* and *connectors*.

Shaw et al. [98] give a definition of a *connector*. Architecturally, a connector is a discrete design element, representing a set of mechanisms that mediate interactions between components. This interaction may take various forms, such as communication, resource contention, and scheduling concerns. At the implementation level, however, Shaw observes that the realization of such a connector is complex and consists of a number of different concrete artifacts. These include code artifacts (application-level code, libraries/stubs and infrastructure services) and non-code artifacts (data/location tables, policy files and formal specification).

The treatment of connectors as first-class entities [95] has come to be valued in software architecture. When component interactions are embodied at the level of architectural design as *connectors*, this enables the system designer to make interactions explicit and easy to identify, to attach semantics, and to capture abstract relations.

2.1.1 Connectors

Here, we survey the means adopted in existing architectural description languages for modeling connectors.

In C2 [101], a style originally intended for systems that have a graphical user interface (GUI), message-passing connectors are used to route, broadcast, and filter messages between architectural layers.

The Acme [47] architecture description language allows hierarchical architectures in which a component or connector may have a representation as a subsystem itself composed on components and connectors. This means of decomposing connectors [99, 100] could be used to support the depiction of "complex" connectors.

One benefit of the treatment of connectors as first-class, as Allen demonstrated with Wright [12],

is to enable their formal specification and analysis, independent of the components they are to connect. The Wright-based formal analysis of the High Level Architecture (HLA) for Distributed Simulation [13], which was successful in revealing interesting flaws in the proposed connector design, also illustrates a concern for how specific connectors can be built up in a traceable and modular way. In Wright, a connector may be specified as a composition of subprocesses.

The Reo [15, 20] approach to software connectors is grounded in dataflow networks and is similar in spirit to hardware design of asynchronous circuits. In the Reo approach, the simplest kind of connector is a "channel", which has a source end and a sink end. Complex connectors are connected as graphs of these channels, where a node in the graph represents a set of one or more channel-ends, and an edge in the graph represents at least one channel. A component can connect to a node only if it is homogeneous: either all source or all sink ends. The semantics are given in terms of timed data streams and constraint automata, enabling checks that determine whether one connector's behavior is identical to, or is a refinement of, another connector's behavior.

At a higher level of abstraction, Medvidovic and Taylor's classification of existing architecture description languages [71], or ADLs, gives a set of features that characterize how connectors are represented within a particular ADL. These features include the extent of support for modeling complex connector types and the support for generating implementations of simple connector instances. They observe that existing ADLs tend to support only one of the two.

2.2 Taxonomy

The feature-based classification for architectural styles given by Shaw and Clements [96] identifies a small set of abstract connectors as a part of a style discrimination framework. Their work provided a basis for further classification efforts that focused specifically on connectors rather than architectural styles.

A "periodic table"-inspired classification by Hirsch et al. [52] proposes a set of properties (such as *Broadcast*, *Reliable*, *Typed* and *Synchronous*) for discriminating between existing connector types. They argue that a means of classification would assist in the definition of operations over connectors and the creation of specialized connector variants that have additional properties.

Mehta [75] present a framework for classifying connector types. It includes four kinds of services provided by connectors (communication, coordination, conversion, facilitation), and eight kinds of connector type (procedure call, data access, linkage, stream, event, arbitrator, adaptor, distributor). They argue that a taxonomy of connectors can help in the process of selecting connectors that are appropriate for a particular system's needs, and, furthermore, an understanding of the relationship of the characteristics of connectors gained from such as taxonomy can be used as an aid in the synthesis of new varieties of connectors.

2.3 Connector Code Generation

2.3.1 Connector Implementations

UniCon [98] addresses implementation issues in realizing specific connectors. The UniCon compiler enables the construction of a system from an architecture description including generation

of the code and other necessary constructs that implement the system's connectors. A specific set of connector abstractions is supported. UniCon focuses on assembling system implementations by generating instances of existing connector types.

Similarly, ArchShell [72], for the C2 architectural style, includes development support for constructing and modifying Java and C++ software systems that use C2-style message-passing connector types. F Prime [31] provides support for specifying the architecture for the flight software for small satellites. It also provides a C++ framework for generating code from the architecture specification.

In their exploration of the use of off-the-shelf middleware to implement C2-style connectors in a distributed software system, Dashofy et al. [40] briefly discuss the possible merits of combining two or more middleware implementations within a single "virtual connector" so that either implementation may be selected.

GenVoca [26] takes a domain-specific approach to generation of component interaction code and illustrates the leverage that can be gained from restriction to a particular domain.

Booch components [32], a reusable component library available for several object-oriented languages including Ada and C++, strives to separate "policy" and "implementation" by providing a collection of abstract things (lists, maps, stacks, etc.) each of which has numerous implementation variants so as to offer programmers a vast array of tradeoffs in efficiency in time and space.

2.4 Connector Evolution

A number of approaches facilitate architectural evolution, particularly architectural refactoring [108]. Barnes et al. [23] introduced the notion of evolution styles to enable architects to develop an evolution plan for changing the architecture of a system in a series of stages. However, the actual change to the architecture still needs to be carried out in the code. Grunske [50] formalizes architectural refactorings as hypergraph transformation rules that can be applied automatically. Ivkovic et al. [53] annotate architectural models with non-functional requirements, then use these and other constraints to select refactoring actions. Pashov et al. [86] use a feature model along with traceability links to architectural elements to generate suggestions for architectural restructuring. However, in these approaches, the architecture specification is not integrated with the code, so transformation of the code has to be carried out manually or with a separate tool. Approaches such as C2-SADEL [73], Darwin [66], ArchWare [83] and Plastik [55] do provide a mapping between an architecture description language (ADL) and a runtime framework in order to implement the changes, but these approaches lack a mechanism to prevent bypassing of architectural abstractions in the code. Moreover, they support only a handful of built-in connectors, while our approach has a generic framework supporting arbitrary connectors.

Architectural transformations may also be carried out by refactoring source code directly [90, 94]. Several automated approaches for refactoring code have been proposed in the literature [89, 92, 105]. Our approach is manual, but the changes are made in the high level architecture specification rather than in the source code, providing better extensibility. Changing connectors by refactoring source code would require mappings to be provided between all pairs of interchangable connectors; thus work grows quadratically in the number of connectors. In our approach, only the implementation of the connector needs to be supplied; one connector can replace another one as

long as the component interfaces are compatible with its requirements. The work required to be done in our approach thus grows linearly in the number of connectors.

Aßmann et al. [17, 18] introduced the use of metaprogramming [65] to generate glue code for connectors. We have extended metaprogramming to additionally perform typechecking on connectors as provided by ArchJava [11].

We use object capabilities for modularizing connector implementations. The object capability model was first introduced to support the secure construction of systems from untrusted components, using the principle of least authority [44, 77, 78, 79]. We use capabilities to limit component code from accessing middleware libraries that can be used for implementing connectors and thus prevent developers from bypassing our connector abstractions.

Connectors are a crosscutting concern [25, 29]. Aspects [58] have been used to modularize crosscutting concerns, but their application is inherently tied to the structure of each application. In contrast, our approach allows connectors to be implemented once and then used in any application.

2.5 Connector Compatibility

Allen and Garlan [12] introduced a notation for specifying the runtime architecture of a system in terms of its constituent components and connectors. In their notation, a component has ports which are used by the component to interact with other components. Also, a connector has roles which are used to specify the expected behavior of the components that would be attached to the connector. The semantics of ports and roles are specified using CSP processes. They define an analysis that can be used to check if a component's port is compatible with a connector's role in terms of communication events in the interaction protocol. This is insufficient for our purposes in this thesis as we are interested in the dataflow behavior of a connector. Although it is possible to perform dataflow analysis of CSP programs [39], we have used constraint automata instead as it allows explicit modeling of the dataflow behavior.

Mehta and Medvidovic [74] provide a connector compatibility matrix based on their taxonomy [75] to determine if two connectors are compatible. Their approach has two limitations. First, the analysis has to be done manually. Second, it is imprecise. For example, when replacing a pipe connector with an event-based connector, their analysis can tell that there are restrictions on cardinality. However, to determine the specific restrictions, one would have to compare the dimension values from their taxonomy.

Chapter 3

Connector Evolution Approach

In this chapter, we present our architecture-centric connector evolution approach. Our approach involves integrating an architecture specification with the code.

3.1 Motivating Example

We illustrate connector evolution issues by considering the example system shown in figure 3.1. While simple, this example serves to illuminate the challenges involved in connector evolution tasks in realistic systems. The figure shows the evolution of a producer-consumer system based on the message broker architectural pattern [22]. Here, the message broker acts as the connector, facilitating interactions between the producer and consumer. Suppose the system was initially implemented using RabbitMQ[1] as the message broker and must be migrated to use Apache Kafka[2] instead. (In real-world systems, this might be justified by the better performance offered by Apache Kafka over RabbitMQ [30]). The migration of the producer is shown in figure 3.1a. In the example system, the producer generates the string `'Hello World!'` and publishes it to the message broker. The left-hand side of figure 3.1a shows the implementation of the producer using RabbitMQ, while the right-hand side shows the implementation based on Apache Kafka after the migration. Similarly, the migration of the consumer is shown in figure 3.1b. The consumer reads the strings generated by the producer and displays them. Changing the connector in this example presents several challenges.

Non-localized changes. First, the change is large and non-local. In the example system, the source code files for both the producer and the consumer need to be changed, as can be seen in figure 3.1. The changes span multiple lines in each file (lines 1-11 and 15 in the producer, and lines 1-14 and 18-20 in the consumer). This issue is exacerbated as the system grows larger: in the limit, such changes might be littered across hundreds of files and thousands of lines across those files—an edit truly tedious to make by hand.

The changes are non-localized because connectors are typically implemented by using middleware (e.g., RabbitMQ or Apache Kafka), and there is tight coupling between the implementation of the system and the middleware used [34, 40, 70, 87]. Generally speaking, the implementation

[1]RabbitMQ. https://www.rabbitmq.com
[2]Apache Kafka. https://kafka.apache.org

```python
import pika

connection = pika.BlockingConnection(
  pika.ConnectionParameters(host='localhost'))

channel = connection.channel()

channel.queue_declare(queue='hello')

channel.basic_publish(exchange='',
  routing_key='hello', body='Hello World!')

print("[x] Sent 'Hello World!'")

connection.close()
```

```python
from kafka import KafkaProducer

producer = KafkaProducer(
  bootstrap_servers=['localhost:9092'])

producer.send('hello', b'Hello World!')

producer.flush()

print("[x] Sent 'Hello World!'")

producer.close()
```

(a) Data producer using RabbitMQ (left) and Apache Kafka (right)

```python
import pika

def callback(ch, method, properties, body):
  print("[x] Received %r" % body)

connection = pika.BlockingConnection(
  pika.ConnectionParameters(host='localhost'))

channel = connection.channel()

channel.queue_declare(queue='hello')

channel.basic_consume(queue='hello',
  on_message_callback=callback, auto_ack=True)

print('[*] Waiting for messages. '
  'To exit, press CTRL+C')
channel.start_consuming()

connection.close()
```

```python
from kafka import KafkaConsumer

consumer = KafkaConsumer('hello',
  bootstrap_servers=['localhost:9092'])

try:
  print('[*] Waiting for messages. '
    'To exit, press CTRL+C')
  while True:
    raw_msgs = consumer.poll(
      timeout_ms=100000)

    for tp, msgs in raw_msgs.items():
      for msg in msgs:
        print('[x] Received %r'
          % msg.value)

finally:
  consumer.close()
```

(b) Data consumer using RabbitMQ (left) and Apache Kafka (right)

Figure 3.1: Migration of a producer-consumer system from RabbitMQ to Apache Kafka

of a connector involves connector initialization, data exchange over the connector and cleanup. For example, in our example system in figure 3.1, we establish a channel by making a connection to the message broker server (connector initialization in green boxes), read and write data over the channel (data exchange in yellow boxes) and tear down the channel by closing the connection to the message broker (connector cleanup in orange boxes). Typically, these operations are spread across the codebase, as data exchange needs to be performed in various system components. The operations depend on middleware libraries, meaning that dependencies on these libraries are also strewn all over the codebase, resulting in tight coupling between the system implementation and the middleware library. Thus, changing the connector leads to changing all locations in the code that depend on the middleware library—again, quite tedious.

Potential for data errors. Second, data errors may be introduced into the system during the migration process—i.e. the data that individual components of the system read and write over the connector may be different before vs. after the migration. This may happen in several ways.

One possibility is that developers might make errors in connector setup. For example, the

mechanics of connection establishment are different between RabbitMQ and Apache Kafka (figure 3.1). For RabbitMQ, an *exchange* might need to be created on the RabbitMQ server and *queues* bound to the exchange. Publishers then connect to the RabbitMQ server and publish messages to an exchange, which delivers them to appropriate queues based on the supplied routing key. (RabbitMQ provides a default exchange which has no name. In figure 3.1a, the default exchange is used for publishing as can be seen in lines 10-11 of the RabbitMQ-based producer.) Consumers then subscribe to be notified of messages sent to particular queues. Apache Kafka, in contrast, supports communication based on *topics* created on the Apache Kafka server. To migrate the system to use Apache Kafka instead of RabbitMQ requires RabbitMQ exchanges and message queues to be mapped to Apache Kafka topics. This mapping requires careful thought. During the migration, developers might misconfigure the components of the system to connect to the wrong topics [106]. This might happen because of developer fatigue when a large number of edits need to be performed, resulting in incorrect message delivery [91].

Another possibility is that the data that is exchanged over the connector may be mangled during the migration. This may happen because of the semantic differences in the data exchange APIs associated with the connectors. For example, in RabbitMQ, the `body` parameter of the `basic_publish` function (used to specify the message to be sent) can be a string. However, in Apache Kafka, the message parameter of the `send` function has to be of type bytes. In the example in figure 3.1, this is achieved by prefixing a `b` to the string literal `'Hello World!'` (line 6 in the Apache Kafka-based producer in figure 3.1a). If the message parameter doesn't have the bytes type, then sending messages would fail. Another way to do this, applicable to non-string messages, is to provide a serializer in the producer and a deserializer in the consumer. However, this leads to the introduction of boilerplate code and might cause type mismatch errors between the serialization and deserialization sides.

Potential for behavioral errors. Third, behavioral errors may be introduced into the system during the migration process, which may involve a large number of complex refactorings. For example, consider the transformation of the consumer. In the RabbitMQ-based consumer, there is a callback function for receiving messages. In the case of Apache Kafka, however, the consumer must poll the topic that it is interested in periodically for retrieving messages from the topic. Making this change would require a fairly involved refactoring. Previous studies have shown that complicated refactorings can potentially introduce bugs in the code [27, 28, 85, 93]. As the size and complexity of the migrated system grows, the potential to introduce errors in refactoring may increase, leading to differences in the behavior of the affected components before and after the migration.

3.2 Goals

Our objective is to support the evolution of connectors, mitigating the issues outlined in the Example section. Guided by these issues, we have drawn up the following goals:

- **Localized changes:** The modifications that need to be made to change the connector must be localized to a few lines in a few files *(mitigates non-localized changes).*
- **Data uniformity:** The data that individual components read from and write to each

connector to must remain the same before and after the connector change *(mitigates potential data errors).*

- **Semantic equivalence of individual components:** There should be no change in the behavior of individual components, provided that the data that each component reads from and writes to each connector remains the same after the connector change *(mitigates potential behavioral errors).*

3.3 Architecture-Centric Development Approach

To develop a connector evolution approach fulfilling the above goals, we first derive requirements from these goals.

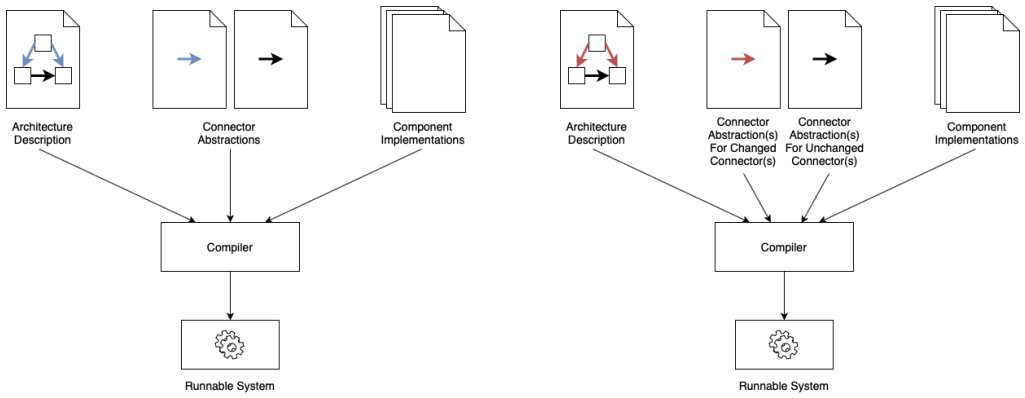
Why are connector implementations scattered across the codebase, resulting in non-localized changes during connector evolution? Existing programming languages and design methodologies do not provide an abstraction mechanism for connectors. As a result, connector implementations are not modularized i.e., there is no locus of definition for a connector that encapsulates its implementation. So, a way to modularize connector implementations must be provided.

Furthermore, even if abstractions for connectors were available, they are only useful if we can ensure that developers use them consistently. In current programming languages, developers have unrestricted access to connector implementation facilities, e.g. language or middleware libraries such as `pika` and `kafka`. Thus, they could potentially implement component interactions directly using those libraries, bypassing connector abstractions. This might be convenient in the moment, yet cause problems later: changing a connector that has been implemented without using the abstraction mechanism may involve edits sprinkled all over the codebase, just as if no abstraction mechanisms for connectors were available. Therefore, developers must be prevented from bypassing connector abstraction mechanisms.

In summary, to meet the goal of localized changes, we require: 1) a way to modularize the implementation of the connector, and 2) a way to prevent developers from bypassing the connector modularization. If these two requirements are met, then the goals of data uniformity and semantic equivalence of components will also be met. For if the connector implementation is modularized in an appropriate manner, then the implementation of components need not be changed at all. Thus, the data that each component reads from and writes to each connector remains the same before and after a connector change has been performed. Consequently, the behavior of each component also remains the same before and after the change.

To meet the requirements stated above, first of all, we provide a way for components to be implemented in such a way that they do not (and cannot) include code for implementing connectors. Our approach, following the object-capability model [79], restricts access to privileged libraries such as those needed to implement connectors, and forbids global state. As a result, components *cannot* interact with other components except through explicit external interfaces. This meets the second requirement laid out above i.e., developers are prevented from bypassing our connector abstractions.

Second, since components are devoid of connector code, a separate, external mechanism must be provided for specifying how components are connected to each other. By separating this specification from component implementations, we ensure connectors can be changed without



(a) Before change (b) After change

Figure 3.2: Changing connectors in our architecture-centric development approach

modifying component code. In our approach, an architecture description language is used to specify the topology of the component connections and the type of connector used for each connection.

Third, we provide an abstraction mechanism that enables the implementation of connectors in a reusable way. The connector abstraction specifies constraints that must be satisfied by the component interfaces to which it connects. Different components may use different operations and data types in their interfaces, and connector implementation code may therefore differ based on these operations and data types. Our connector abstractions therefore specify an approach to generating this code that is specific to the connector but generic over the component interfaces connected, thus enabling the reuse of the connector abstraction. In other words, a connector abstraction can be used to implement connectors that have the same conceptual semantics (e.g., synchrony/asynchrony, reliability and performance characteristics, etc.) but might involve different operations and data types. This fulfills the first requirement mentioned above, i.e. provision of a mechanism for connector modularization.

Our approach is illustrated in figure 3.2. The figure shows the artifacts that must be provided for implementing a system: developers must provide an architecture specification, implementations of all the components in the architecture, and connector abstractions for all the connectors used. These three sets of artifacts are compiled to generate the executable code for the system.

As seen in figure 3.2b, changing a connector involves simply changing the architecture description to specify the new connector to be used and supplying the definition of the connector. As mentioned above, connector definitions are reusable. If the definition of the connector is already available, all changes required to be made to switch connectors are, in effect, localized to the architecture description.

Further, the goals of data uniformity and semantic equivalence of components are also satisfied by our approach provided that both the architecture description and the connector implementation are correct. This is because the code for implementing components is not changed at all when connectors are changed. Consequently, if the architecture description and the connector implementation are correct, then the data that each component reads from and writes to each connector remains the same before and after the connector change. Moreover, the behavior of

the components also remains the same before and after the change. To verify that the connector implementation is correct i.e., data is transferred through the connector without any errors—we only have to check the definition for the connector, a task that is significantly easier than checking connector-related code strewn across the codebase in conventional approaches.

Thus, overall, our approach consists of two main facets: 1) an explicit specification of the architecture description of the system, and 2) integration of the architecture description with the implementation of the system. We will now explain these two facets in detail.

3.3.1 Architecture Description

To allow developers to specify software architectures, we provide language constructs for declaring *component types and instances*, *ports*, *connector types and instances*, and *attachments*. These constructs are shown in figure 3.3 which presents the architecture description of the RabbitMQ-based producer-consumer system that was introduced in section 3.1.

Component types and Ports. A component type describes the structure of a component. It may have one or more ports, which are points of interaction with other components. Ports have an associated interface, which is a collection of one or more methods. Port interfaces may be of one of two types - `provides` or `requires`. The methods in a `provides` port interface are implemented by the component and can be called by other components that are connected to this port. On the other hand, the methods in a `requires` port interface must be provided by some other component connected to this port. In figure 3.3, we have two component types - `Producer` (lines 1-2) and `Consumer` (lines 4-5). The `Producer` component type has a `requires` port named `data_pub` (line 2) and the `Consumer` component type has a `provides` port named `data_sub` (line 5). The interface for both these ports is `StringDataIface`, whose definition is shown at the bottom of the figure. It contains a single method named `processData` which takes a `String` argument and returns `Unit` (i.e. void).

Connector types. A connector type represents a specific mechanism of interaction between components. A connector type may have associated attributes which must be properly configured to set up the interaction between the connected components correctly. In figure 3.3, we have a connector type named `RabbitMQQueue` that represents interaction based on RabbitMQ message queues (lines 7-9). This connector type has two attributes: 1) a `String`-valued attribute named `host` which is used to configure the hostname or the IP address of the RabbitMQ server, and 2) a `String`-valued attribute named `name` which is used to configure the queue name.

Component instances. Component instances are instantiations of component types. Each instance of a component type corresponds to a different copy of the component at runtime. In the architecture description, component instances are listed in a `components` block within an `architecture` block. This is shown in lines 12-14 in figure 3.3. In the figure, we have two component instances - an instance named `p` of the component type `Producer` and an instance named `c` of the component type `Consumer`.

Connector instances. Similar to component instances, connector instances are instantiations of connector types. In the architecture description, connector instances are listed in a `connectors` block under the `architecture` block. This is shown in lines 16-17 in figure 3.3. In the figure, we declare an instance named `q` of the connector type `RabbitMQQueue`. If a connector type has attributes associated with it, the values for these attributes can be supplied

```
1  component Producer
2      port data_pub: requires StringDataIface
3
4  component Consumer
5      port data_sub: provides StringDataIface
6
7  connector RabbitMQQueue
8      val host: String
9      val name: String
10
11 architecture
12     components
13         Producer p
14         Consumer c
15
16     connectors
17         RabbitMQQueue q with host='localhost' and name='hello'
18
19     attachments
20         connect p.data_pub and c.data_sub with q
```

```
1  type StringDataIface
2      def processData(s: String): Unit
```

Figure 3.3: Architecture description of the RabbitMQ-based producer-consumer system

when a connector instance is declared. As shown in line 17 in figure 3.3, the values of the attributes `host` and `name` for the connector instance `q` have been set to `'localhost'` and `'hello'`, respectively.

Attachments. An attachment specifies which ports of which components are connected together by which connector. Attachments are spelled out in an `attachments` block within the `architecture` block. As shown in lines 19-20 of figure 3.3, attachments are specified using the `connect` keyword. In line 20, the `data_pub` port of the `Producer` component instance `p` and the `data_sub` port of the `Consumer` component instance `c` are being connected together using the `RabbitMQQueue` connector instance `q`.

3.3.2 Integration of Architecture Description with Code

To implement a system, developers must supplement the architecture specification with implementations of components as well as connector abstractions. Component implementations focus only on functionality, while connector abstractions generate code that links components together via their port interfaces. In our approach, the compiler instantiates components and the generated connectors to link the system together according to the architecture specification.

For the implementation language, we use WYVERN[3], a programming language we designed with a built-in object-capability mechanism to prevent developers from bypassing our connector

[3]An early prototype is available at https://github.com/selvasamuel/wyvern.

```
1 module def Producer(data_pub: StringDataIface)
2     ...
3     data_pub.processData('Hello World!')
4     ...
```

```
1 module def Consumer()
2     ...
3     val data_sub: StringDataIface = new
4         def processData(s: String): Unit
5             ...
```

Figure 3.4: Module definitions for the `Producer` and `Consumer` component types

modularization.

Component Implementations and Port Reifications. Components are implemented by means of modules. In WYVERN, modules are functors à la ML [103]. In other words, modules are functions that accept zero or more arguments, each of which is a module instance of a specified type, and produce a module instance as a result. There must be a module definition for each component type in the architecture description. The module definitions are matched to the respective component types by name. The module definitions for the two component types - `Producer` and `Consumer` - in the architecture description in figure 3.3 are shown in figure 3.4.

Requires ports are reified via arguments in module definitions. Requires ports and the corresponding module definition arguments must have the same name and the same type. This is shown in the definition of the `Producer` module in figure 3.4. Corresponding to the `data_pub` port of the component type `Producer`, which is a requires port, we have an argument named `data_pub` in the definition of the `Producer` module. Moreover, the `data_pub` argument has the type `StringDataIface`, which is the same type as that of the `data_pub` port. The typechecker in WYVERN ensures that the order of the arguments in module definitions is the same as the order of the ports declared in the corresponding component types in the architecture description.

Provides ports are reified by fields in module definitions; at run time, the fields hold objects of the respective port type. As in the case of requires ports, provides ports and the corresponding fields of module definitions must have the same name and the same type. This can be seen in the definition of the `Consumer` module in figure 3.4. The `Consumer` component type has a provides port named `data_sub`, which has the type `StringDataIface`. This port is reified by the field `data_sub` shown in line 3 of the definition of the module `Consumer` in figure 3.4. The `data_sub` field also has the type `StringDataIface`. The value of the field is set to an object which provides an implementation of the methods in its type. In this case, the object implements the `processData` method in the type `StringDataIface` which is the only method in the type (lines 3-5 in the definition of `Consumer` in figure 3.4).

Connector Abstractions. Connector abstractions are implemented via types. Corresponding to each connector type in the architecture description, there must be a type definition which modularizes the connector's implementation. The type definition for the `RabbitMQQueue` connector type in the architecture description in figure 3.3 is shown in figure 3.5. The name of

```
1  type RabbitMQProperties
2      val host: String
3      val name: String
4      metadata new
5          def checkPortCompatibility(l: list[ast.PortDecl]): Boolean
6              ...
7
8          def generateConnectorImpl(portInterface: ast.AST): list[ast.AST]
9              ...
10
11         def generateConnectorInit(c: list[ast.ComponentInst],
               ↪ p: list[ast.PortDecl], ctrName: String,
               ↪ ctrProps: RabbitMQQueueProperties): list[ast.AST]
12             ...
```

Figure 3.5: Type definition corresponding to the `RabbitMQQueue` connector type

the type is derived from the name of the connector type by adding the suffix `-Properties`, e.g. `RabbitMQQueueProperties` in figure 3.5. The type must have the same attributes, of the same types, as the respective connector type (e.g. `host` and `name` of type `String` in the connector implementation type `RabbitMQQueueProperties`.

Connector modularization is achieved by using type-based metaprogramming. In WYVERN, metadata can be added to types. The metadata consists of a set of methods that can be executed at compile-time. In other words, methods can be defined in the metadata of a type to be executed by the compiler to perform various tasks while processing values of the type at compilation time, such as custom typechecking [42], custom type conversion, etc. For connector abstraction, we use metadata methods in the connector implementation type for generating the code for implementing connectors.

We have predefined a set of three metadata methods that are implemented for each connector. These methods can be seen in the definition of the `RabbitMQQueueProperties` type (lines 4-12) in figure 3.5. The methods are executed by the compiler during the processing of each `connect` statement in the `attachments` section in the architecture description (see lines 19-20 in figure 3.3).

First, the method `checkPortCompatibility` is used to implement checks to ensure that the ports tied together by a connector are compatible with each other. For example, a custom check may be implemented to ensure that all the ports involved in a connection have the same interface. The semantics of compatibility of ports can vary based on the connector. For example, in the case of a connector based on the publish/subscribe paradigm, the method in the port interface is checked to have return type `Unit`, as the publisher does not expect a return value from the subscriber(s). The required typechecking semantics for each connector can be implemented in this method.

Second, the method `generateConnectorImpl` is used to specify code that should be generated to implement a connector's functionality. Specifically, the code that should be generated to implement the operations that must be performed by the `requires` and `provides` ports to enable component interaction via the connector is specified in this method. For example,

```
1 module def RabbitMQQueue_StringDataIface_requires(
      ↪ props: RabbitMQQueueProperties, rmq: LibRabbitMQ): StringDataIface
2      ...
3      def processData(s: String): Unit
4          ...
```

Figure 3.6: Module generated for the `requires` port with `StringDataIface` interface connected by a `RabbitMQQueue` connector

the module that is generated for the `requires` port of the `Producer` component type is shown in figure 3.6. The generated modules may have arguments that accept capabilities to the middleware libraries that support connector implementation; these capabilities are the only way that a module in WYVERN can use operating system communication facilities [76]. For example, the module `RabbitMQ_StringDataIface_requires` in figure 3.6 takes an argument of type `LibRabbitMQ`, representing a library module that supports implementation of interactions via RabbitMQ message queues.

Third, the method `generateConnectorInit` is used to specify code that should be generated to create and initialize instances of modules that implement the ports involved in a particular connector. Note that the modules that implement the ports are the ones that are generated by the `generateConnectorImpl` method. To instantiate port modules, appropriate modules that have to be passed as arguments must be instantiated and they must be passed in the required order. In general, the compiler cannot make any assumptions about the modules that port implementation modules can depend on or the order of these dependencies in their signatures. As a result, the `generateConnectorInit` method has been provided for connector implementers to specify how the port modules should be instantiated.

Executable System Generation. A perceptive reader might have noticed that we require only definitions of the modules implementing components to be provided. To create an executable system, appropriate instances of these modules must be created. The code for doing this is automatically generated by the compiler based on the declared component instances in the architecture description. To create an instance for a module corresponding to a component, the compiler uses the code generated from the `generateConnectorInit` method above to initialize the `requires` ports and uses them as arguments to instantiate the modules for the components. It also links the `provides` ports of the component to the connectors that use them. Finally, it calls the `main` method of the module to start the component.

Preventing Connector Abstraction Bypass. As noted above, the modules that provide access to middleware libraries for connector implementations are object capabilities that need to be explicitly passed as arguments to modules that require them. In our approach, the generated connector code modules can freely access any middleware library that they require. However, modules that implement components cannot access them. This is enforced by not having these capabilities in the arguments in module signatures. For example, the `Producer` and `Consumer` modules in figure 3.4 do not have access to the RabbitMQ library module. So, the RabbitMQ connector cannot be used in those modules. Moreover, since the compiler is responsible for creating instances of the modules implementing components, this restriction cannot be bypassed

```
...
connector ApacheKafkaTopic
    val host: String
    val name: String

architecture
    ...
    connectors
        ApacheKafkaTopic t with host='localhost' and name='hello'

    attachments
        connect p.data_pub and c.data_sub with t
    ...
```

```
type ApacheKafkaTopicProperties
    val host: String
    val name: String
    metadata new
        ...
```

Figure 3.7: Architecture description and connector abstraction for Apache Kafka-based producer-consumer system

by developers, thus forcing them to use the provided connector abstractions.

3.3.3 Connector Modification

To change a connector, the corresponding connector type and connector instances in the architecture description have to be changed. Moreover, the connector abstraction that implements the new connector has to be supplied as well. For example, the changes that need to be made to the architecture description in figure 3.3 to change the `RabbitMQQueue` connector to `ApacheKafkaTopic` is shown in figure 3.7. The figure also shows the type `ApacheKafkaTopicProperties` that implements the connector abstraction and which must additionally be provided. As can be seen, the edits for switching to a new connector are localized to the architecture description, thus fulfilling our goals outlined in section 3.2.

Chapter 4

Connector Compatibility Analysis

4.1 Connector Compatibility

Suppose a connector in a software system is being replaced with another connector. The replacement connector is said to be *compatible* with the connector being replaced with respect to a certain behavioral dimension if the behavior of the replacement connector along that dimension subsumes that of the connector being replaced. For example, suppose an event-based connector is being replaced with a different event-based connector. Let's say the behavioral dimension we're interested in for these two connectors is the delivery policy they use. Suppose that the connector being replaced uses a best effort delivery policy. In other words, an event notification is sent to a recipient just once. The recipient may not receive the notification if there are failures. Moreover, suppose that the replacement connector uses a guaranteed delivery policy. This means that it is ensured that a recipient of an event receives a notification about the event even in the face of failures. In this case, the replacement connector is compatible with the connector being replaced because every event notification that would reach a recipient when the connector being replaced is used would also reach the recipient if the replacement connector is used instead. On the other hand, if the two connectors are interchanged, then the replacement connector (i.e., the connector with the best effort delivery policy) would be incompatible with the connector being replaced (i.e., the connector with the guaranteed delivery policy). This is because some event notifications would be lost if the replacement connector is used but no event notification would be lost if the connector being replaced is used.

When changing the connectors in a given software system, the architects might come up with a set of candidate connectors that could satisfy the new requirements. Even though one or more connectors are being replaced, some behaviors of the original connectors might need to be preserved even after the change has been made. Thus, the architects might be interested in knowing which of the candidate connectors are semantically compatible with the connector to be changed with respect to the behaviors of interest that need to be preserved. In particular, given the specifications of two connectors, we would like to determine if they are compatible with each other.

Compatibility could be assessed along several dimensions. For example, one important dimension for which architects might be interested in checking compatibility is with respect to

real-time constraints. There might be real-time constraints on the delivery of data that is enforced by the connector(s) being replaced. The architects might be interested in assessing whether those constraints would be met by the replacement connector(s) as well.

In this thesis, we focus on *data relay compatibility*. By data relay compatibility, we mean that the replacement connector should be able to accept all data coming in from input ports that the connector being replaced can accept. Moreover, whenever an input data item is delivered to an output port by the original connector, the replacement connector should deliver it to a corresponding output port as well. Note that ensuring data relay compatibility is integral to achieving the goal of data uniformity in our connector evolution approach (see section 3.2).

Checking connector compatibility can be a challenging task. We will illustrate this with three scenarios. First, consider two connectors each of which can be connected to a single input port of a component and a single output port of another component. Further, suppose that both connectors accept only integers as input as well as output only integers. Now, let's say the first connector simply relays each integer input to the output but the second connector adds 5 to the input integer and delivers the sum to the output. It is clear that the two connectors are incompatible even though their input and output interfaces are the same. We desire to enable architects to determine this without examining the code for the connectors.

Second, consider two connectors each of which can again be attached to a single input port and a single output port. As in the previous case, suppose that the connectors input and output only integers. Suppose that the first connector simply relays each incoming integer to the output in a single step. On the other hand, let's say that the second connector relays incoming integers to the output in two steps; it first stores an incoming integer in a memory cell and then reads out the memory cell to deliver the value to the output. These two connectors are compatible with each other from the standpoint of their data relay functionality. Again, we want to enable architects to determine this without having to examine the code for the connectors which can be challenging for more complicated connectors.

Third, for a more realistic scenario, suppose we have the implementation of some functionality for a robot available as a ROS 2 package. ROS 2 (Robot Operating System 2) [7] is an open source framework for building robotic software, providing a collection of libraries and tools for a wide variety of robotic platforms. ROS 2-based systems are structured as independent components called *nodes*, which communicate with each other using a variety of communication primitives (such as *topics* which are a publish/subscribe mechanism, *services* which are an RPC mechanism, etc.). Suppose that in our package, the nodes communicate using services. Now, not all robotic platforms have good support for ROS 2. For example, ROS 2 is poorly supported on the Spot robot from Boston Dynamics [8]. So, to use the ROS 2 package with Spot, the RPC-based ROS 2 service connectors have to be replaced with another connector. Usually, gRPC, which is also an RPC implementation, is used for communication between components running on the Spot robot [9]. Now, ROS 2 systems can be configured for best effort delivery (in which case messages are sent at most once and may be lost if there are failures) or for guaranteed delivery (which ensures that a recipient receives a message that is sent to it even in the face of failures). These options are configured in the code using the API provided by ROS 2. Previous studies have found that such architectural knowledge implicitly specified in the code can get lost over time [45]. As a result, the architects might not remember the delivery policy used in the ROS 2 implementation. So, when the package is changed to use gRPC, if the original implementation used guaranteed

delivery, then the migrated package might not function correctly as gRPC provides best effort delivery. This error might not be detected until after the migration has been completed and it might be costly to fix at that time. A mechanism for early detection of such incompatibilities between connectors without the need for examining the code would be helpful for architects in planning the migration process.

4.2 Connector Compatibility Analysis Approach

We have developed an extensible mechanism for performing connector compatibility analysis. The mechanism involves specifying the relevant semantics for checking compatibility along a dimension of interest in the connector abstraction. In this thesis, we restrict ourselves to checking data relay compatibility. However, the mechanism can be extended to check compatibility along other dimensions. For specifying the data relay semantics of a connector for the purpose of checking data relay compatibility, we extend the constraint automata formalism that was introduced to specify the semantics of Reo connectors [20, 54].

4.2.1 Constraint Automata

A constraint automaton is a labelled transition system in which the nodes represent the states that a connector can be in and the edges represent the state transitions that are possible. The edges are labelled with the set of ports that participate in the interaction represented by the edge, as well as a boolean condition that must be true when that interaction occurs.

Before we provide a formal definition of a constraint automaton, we must set the stage with some preliminary definitions.

Definition 1 (Data)**.** $\mathfrak{D}$ *denotes the set of all data, ranged over by* d*.*

The elements present in the set $\mathfrak{D}$ depend on the use case. If our goal is code generation for the connector, for example, $\mathfrak{D}$ may be infinite and may contain all objects of the data type specified in the port definition. For verification, on the other hand, $\mathfrak{D}$ may be defined as a small, finite set of values. All further definitions work irrespective of whether $\mathfrak{D}$ is finite or infinite, countably or otherwise.

Definition 2 (Port)**.** $\mathfrak{P}$ *denotes the set of all ports in a connector, ranged over by* p*.*

For the purposes of checking data relay compatibility, a connector can be thought of as a set of channels linked to one another. The channel endpoints are called ports. Ports that can be connected to components can be input, output or input/output ports depending on whether they can receive or send data. Endpoints that are common between two channels are internal ports.

Definition 3 (Memory cell)**.** $\mathfrak{M}$ *denotes the set of all memory cells, ranged over by* m*.*

Out of ports and memory cells, we construct *data variables*. We do this to enable modeling of data passing through ports as well as data reads and writes involving memory cells.

Every data variable models a conduit for data. For instance, data can come in at a port or be transmitted out of it. So, we will have a data variable corresponding to every port. Similarly, data can be read from memory cells or written into them. However, contrary to ports, the data in memory cells is persistent and we need to distinguish between the value of a memory cell *before* an execution step and that *after* it. So, a memory cell before an execution step has a different

identity from the same memory cell after that step and we will use different data variables to represent the two cases. Borrowing notation from Petri nets, for every memory cell m, we will have two data variables: $\bullet m$ and $m\bullet$. The data variable $\bullet m$ refers to the data value in m before an execution step, while $m\bullet$ refers to the data value in m after that execution step. Suppose M is a set of memory cells. Then, we abbreviate the sets $\{\bullet m \mid m \in M\}$ and $\{m\bullet \mid m \in M\}$ as $\bullet M$ and $M\bullet$.

Definition 4 (Data variable)**.** *A data variable is an entity x generated by the following grammar:*

$$x ::= p \mid \bullet m \mid m\bullet \qquad \textit{(data variables)}$$

where p and m range over ports and memory cells, respectively. $\mathfrak{X}$ denotes the set of all data variables.

Data variables can be assigned meaning with *data assignments*.

Definition 5 (Data assignment)**.** *A data assignment is a partial function from data variables to data values. $\mathbb{A}\text{SSIGNM} = \mathfrak{X} \rightharpoonup \mathfrak{D}$ denotes the set of all data assignments, ranged over by σ.*

Essentially, a data assignment σ models an execution step involving the ports and memory cells in $\text{Dom}(\sigma)$ and the data values in $\text{Img}(\sigma)$. For instance, $\{p_1 \mapsto \texttt{0}, p_2 \mapsto \texttt{0}\}$ can model an execution step where data value $\texttt{0}$ flows from port p_1 to port p_2, $\{p_1 \mapsto \texttt{0}, m\bullet \mapsto \texttt{0}\}$ can model an execution step where $\texttt{0}$ flows from p_1 into memory cell m, while $\{p_2 \mapsto \texttt{0}, \bullet m \mapsto \texttt{0}\}$ can model an execution step where $\texttt{0}$ flows out of m to p_2. As shown in these examples, data assignments do not capture the *direction* of data-flow: each data-flow can be modeled by a data assignment, but a particular data assignment may model multiple data-flows, depending on directions. For instance, the first example above $\{p_1 \mapsto \texttt{0}, p_2 \mapsto \texttt{0}\}$ may model $\texttt{0}$ flowing from p_1 to p_2 or from p_2 to p_1. As a result, data assignments are *declarative* specifications of data relay behavior; they merely indicate *what* happens (such-and-such data item passes through so-and-so port), not *how* it happens (such-and-such data item is input, or alternatively output, at so-and-so port). Below, when we formally define a constraint automaton, we will take a more *imperative* approach by tagging ports as input or output.

Out of data variables, we can construct data constraints. Suppose that M is a subset of $\mathfrak{M}$.

Definition 6 (Data constraint)**.** *A data constraint is an entity ϕ generated by the following grammar:*

$$a ::= \bot \mid \top \mid x = x \mid \texttt{Keep}(M) \qquad \textit{(data atoms)}$$

$$\ell ::= a \mid \neg a \qquad \textit{(data literals)}$$

$$\phi ::= \ell \mid \exists x.\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \qquad \textit{(data constraints)}$$

$\mathfrak{DC}$ denotes the set of all data constraints.

The atom $\bot$ represents the boolean predicate that is identically false. Similarly, $\top$ represents the boolean predicate that is identically true. The atom $\texttt{Keep}(M)$ is a syntactic sugar for the predicate $\forall m \in M.\ \bullet m = m\bullet$. In other words, it expresses the fact that each memory cell in M retains its value during an execution step.

As a shorthand, we will use $\bigwedge \Phi$ to denote the conjunction of the data constraints in Φ, and $\bigvee \Phi$ to denote their disjunction. (This notation is well-defined modulo associativity and commutativity of conjunction and disjunction.)

Now, we are ready to define a constraint automaton. We formally define a constraint automaton as a tuple consisting of a set of states Q, a triple of three sets of ports (P^{all}, P^{in}, P^{out}), a set of memory cells M, a transition relation $\longrightarrow$, and a set of initial states Q^0. Set P^{all} contains all ports of the connector being modeled, while P^{in} and P^{out} contain the ports that allow input and the ports that allow output, respectively. Although P^{all} contains the union of P^{in} and P^{out}, the converse does not necessarily hold true: besides input and output ports, P^{all} may also contain internal ports. Alternatively, one can use an explicit set of internal ports P^{int} instead of P^{all}.

Definition 7 (State)**.** *$\mathfrak{Q}$ denotes the set of all states, ranged over by q.*

Let 2^X denote the power set of some set X.

Definition 8 (Constraint automaton)**.** *A constraint automaton is a tuple $(Q, (P^{all}, P^{in}, P^{out}), M, \longrightarrow, Q^0)$ where*

- $Q \subseteq \mathfrak{Q}$
- $(P^{all}, P^{in}, P^{out}) \in 2^{\mathfrak{P}} \times 2^{\mathfrak{P}} \times 2^{\mathfrak{P}}$ *such that* $[P^{in} \cup P^{out} \subseteq P^{all}]$
- $M \subseteq \mathfrak{M}$
- $\longrightarrow \subseteq Q \times 2^{P^{all}} \times \mathfrak{DC} \times Q$ *such that* $q \xrightarrow{P,\phi} q'$ *implies Free(ϕ)* $\subseteq P \cup {}^\bullet M \cup M^\bullet$ *for all* q, q', P, ϕ
- $Q^0 \subseteq Q$

AUTOM *denotes the set of all constraint automata, ranged over by α.*

The requirement Free(ϕ) $\subseteq P \cup {}^\bullet M \cup M^\bullet$ means that the effect of a transition remains local to its own scope: a transition cannot affect or be affected by ports outside its data constraint.

Given a transition $q \xrightarrow{P,\phi} q'$, we use ${}^\bullet M_\phi$ to denote the set of all $m \in M$ that syntactically appear as ${}^\bullet m$ in the data constraint ϕ. Similarly, $M_\phi{}^\bullet$ denotes the set of all $m \in M$ that syntactically appear as $m^\bullet$ in ϕ.

We define a *valuation function* $V_q : M \to \mathfrak{D}$ to designate the value $V_q(m)$ of a memory cell $m \in M$ in a state $q \in Q$, where $V_{q_0}(m) = \emptyset$ for all $m \in M$. A constraint automaton can make a transition $q \xrightarrow{P,\phi} q'$ only if there exists a substitution for every syntactic element p, ${}^\bullet m$ and $m^\bullet$ that appears in ϕ to make it true. A substitution simultaneously replaces every occurrence of p with the data value to be exchanged through the port $p \in P$, every occurrence of ${}^\bullet m$ with a value $v = V_q(m)$ and every occurrence of $m^\bullet$ with a value $v \in \mathfrak{D}$. The valuation function $V_{q'}$ for the target state q' of the transition can be defined as follows: for every $m \in M_\phi{}^\bullet$, $V_{q'}(m)$ is a value $v \in \mathfrak{D}$ whose replacement in ϕ yields a substitution that makes ϕ true; for every $m \in M \setminus M_\phi{}^\bullet$, $V_{q'}(m) = V_q(m)$.

Informally, the operational behavior of a constraint automaton is as follows. It starts in an initial state $q_0 \in Q^0$. In any current state q, the automaton waits until data items arrive at some of the input/output ports $p_i \in P^{all}$. Suppose $P \subseteq P^{all}$. Suppose that a data item arrives at each port $p \in P$. This triggers the automaton to check the data constraints of the outgoing transitions of the current state q and choose a transition $q \xrightarrow{P,\phi} q'$ for which the data constraint ϕ is true with the values of the data items that are in the ports in P as well as the values in the memory cells. If there is no transition from the current state q whose data constraint is satisfied, then nothing happens; no transition is fired and the constraint automaton remains in the current state q.

4.2.2 Symbolic Execution of Constraint Automata

To check data relay compatibility between two connectors, we need to determine the relationship between symbolic input and output values of the connectors. For this, we will perform symbolic execution [59] of the constraint automata corresponding to the two connectors.

In constraint automata, instead of specific data values, we already have a symbolic representation of input and output values. Moreover, the data constraints on the transitions show the possible relations among these symbolically represented data values. This makes constraint automata amenable to symbolic execution.

We need to obtain a symbolic representation of the possible relations between output and input value. Usually, a path-based analysis is used in symbolic execution [21]. A constraint automaton can be treated as a directed graph, wherein a path is defined as usual for graphs. To enumerate the possible execution paths, the symbolic execution tree for a constraint automaton has to be generated. We obtain the regular expression of the constraint automaton to represent its execution paths (and thus the symbolic execution tree) succinctly.

Encoding of Data Values. The symbolic execution tree is formed by traversing the constraint automaton. We start from the initial state in the constraint automaton and walk through all possible paths in the constraint automaton. While traversing a transition $q \xrightarrow{P,\phi} q'$ of the constraint automaton, we record its port set P and its guard ϕ on its corresponding edge in the traversal tree.

As we construct the tree, every time we see a port on a transition, it means that a new data element is observed in the stream of data passing through that port. For a port name p, we use $\tilde{p}$ to denote the (finite) data stream that has passed through p. $\tilde{p}$ is defined as follows:

$\tilde{p} = (p_0, p_1, \ldots, p_{n-1}, p_n)$ where $p_i \in \mathfrak{D}$ for all i such that $0 \leq i \leq n$.

To write the data constraints for each state in the traversal tree, we index the elements of these (finite) data streams $\tilde{p} = (p_0, p_1, \ldots, p_{n-1}, p_n)$ backwards starting from the last element. For the set of all finite streams $FStreams$ over a set of elements $\mathfrak{D}$, we define $last : FStreams \times \mathbb{N}^- \cup \{0\} \rightarrow \mathfrak{D}$, where $\mathbb{N}^-$ is the set of negative integers, as a function such that $last(\tilde{p}, i)$ takes a finite stream $\tilde{p}$ and an integer $i \leq 0$ and returns the ith last element of $\tilde{p}$. Thus, $last(\tilde{p}, 0)$ is the last element of the stream $\tilde{p}$, $last(\tilde{p}, -1)$ is the first element before the last element of $\tilde{p}$, etc. Using the function $last$ on streams, we can rewrite a stream using negative indices, going backward from its last element, as: $\tilde{p} = (last(\tilde{p}, -n), last(\tilde{p}, -(n-1)), \ldots, last(\tilde{p}, -1), last(\tilde{p}, 0))$. For convenience, we use a superscript notation to refer to the function $last$ and place its second argument as a subscript index, writing p_i^ℓ instead of $last(\tilde{p}, i)$. Thus, we write the stream $\tilde{p}$ as $\tilde{p} = (p_{-n}^\ell, p_{-(n-1)}^\ell, \ldots, p_{-1}^\ell, p^\ell)$, where we drop the subscript 0 on the last term p_0^ℓ. We treat the sequence of values assigned to each memory cell as a finite stream and backward-index it accordingly.

Algorithm. We can compute the symbolic output values of a constraint automaton in three steps:

1. Obtain the regular expression for the constraint automaton.
2. Unfold loops in the constraint automaton by expanding its regular expression to obtain the unfolded instance of the regular expression.

(a) (b)

Figure 4.1: An example connector and its constraint automaton

3. Traverse the unfolded instance of the regular expression to build the data streams for the ports and memory cells, and determine the relationship between the input and output data.

We will now explain each of these steps in detail. We will illustrate the steps by using the constraint automaton shown in figure 4.1b as an example. This constraint automaton specifies the behavior of the connector shown in figure 4.1a. This connector has one input port named A and two output ports named B and C. It also has a memory cell, M. As can be seen in figure 4.1b, the constraint automaton for the connector has two states - 1 and 2. State 1 is the initial state. In figure 4.1b, this is indicated by the unlabeled arrow at the left of state 1 pointing to it. We will assume that the connector inputs and outputs only integers.

A data input from port A is first stored in the memory cell M. This is shown in the constraint automaton by the transition from state 1 to state 2. This transition carries the label "$\{A\}, m\bullet = a$". The set $\{A\}$ indicates the ports involved in this transition (just A in this case). The data constraint $m\bullet = a$ indicates that the value of the memory cell M after the transition is the data input from port A.

Once the input data has been stored in the memory cell M, the connector outputs the value through port B or port C depending on whether it is non-negative or negative. If the value is non-negative, the content of memory cell M is output through port B. This is indicated by the upper transition from state 2 to state 1 in figure 4.1b. Again, the set $\{B\}$ in the label of the transition indicates the only port - port B - involved in this transition and the data constraint indicates that the value in the memory cell M is non-negative ($\bullet m \geq 0$) and that it is sent to port B ($b = \bullet m$). If, on the other hand, the value in the memory cell M is negative, it is output through port C. This is indicated by the lower transition from state 2 to state 1 in figure 4.1b.

Step 1: Obtaining the regular expression. As noted above, we use the regular expression of a constraint automaton to represent the symbolic execution tree compactly. We can use Brzozowski's algebraic method [35] combined with Arden's theorem [16] to generate the regular expression for a constraint automaton. To do this, we associate a variable with each state to represent the regular expression that would be obtained if that state were the initial state. We set up a system of equations involving these variables and then solve the system of equations for the variable associated with the initial state.

Setting up the system of equations is straightforward. To each state $q_i \in Q$, we associate a variable R_{q_i}. The equation for R_{q_i} is a union of terms. Each term can be constructed as follows: for a transition $q_i \xrightarrow{P,\phi} q_j$ from q_i to q_j, the term $P[\phi]R_{q_j}$ is added. This leads to a system of

equations of the form:

$$
\begin{aligned}
R_{q_0} &= l_{00}R_{q_0} + l_{01}R_{q_1} + l_{02}R_{q_2} + \dots \\
R_{q_1} &= l_{10}R_{q_0} + l_{11}R_{q_1} + l_{12}R_{q_2} + \dots \\
R_{q_2} &= l_{20}R_{q_0} + l_{21}R_{q_1} + l_{22}R_{q_2} + \dots \\
&\dots \\
R_{q_n} &= l_{n0}R_{q_0} + l_{n1}R_{q_1} + l_{n2}R_{q_2} + \dots
\end{aligned}
$$

Here, q_0 is the initial state and the term $l_{ij}R_{q_j}$ appears on the right-hand side of the equation for R_{q_i} if and only if there is a transition from q_i to q_j with l_{ij} as its label.

As can be seen above, the equations in this system of equations can be recursive. To solve such recursive equations, we can use Arden's theorem [16]. Arden's theorem states that the solution of an equation of the form $R_{q_i} = AR_{q_i} + B$, where A and B are regular expressions, is given by $R_{q_i} = A * B$. Arden's theorem assumes that A doesn't contain the empty string, ϵ. This would always hold in our case because of the use of data constraints in labels of transitions. Also, note that the solution to an equation of the form $R_{q_i} = AR_{q_i}$ is simply $R_{q_i} = A*$.

For the constraint automaton in figure 4.1b, we will obtain the following system of equations:

$$
\begin{aligned}
R_1 &= \{A\}[m\bullet = a]R_2 \\
R_2 &= \{B\}[\bullet m \geq 0, b = \bullet m]R_1 + \{C\}[\bullet m < 0, c = \bullet m]R_1
\end{aligned}
$$

To solve this, we first substitute the expression for R_2 in the right hand side of the equation for R_1 to obtain $R_1 = (\{A\}[m\bullet = a]\{B\}[\bullet m \geq 0, b = \bullet m] + \{A\}[m\bullet = a]\{C\}[\bullet m < 0, c = \bullet m])R_1$. Solving this equation would then give $(\{A\}[m\bullet = a]\{B\}[\bullet m \geq 0, b = \bullet m] + \{A\}[m\bullet = a]\{C\}[\bullet m < 0, c = \bullet m])*$ as the regular expression for the constraint automaton.

Step 2: Unfolding loops. A constraint automaton α imposes a (generally infinite) relation among the data elements that pass through the ports of the corresponding connector and/or are stored in its memory cells. In symbolic execution, what is interesting about this relation is not so much the specific data values that it inter-relates, but the relative position of each such value in its respective stream. In other words, we are interested in the relation $R_\alpha \subseteq \mathfrak{L} \times \mathbb{N} \times \mathfrak{L} \times \mathbb{N}$, where $\mathfrak{L} = M \cup P^{all}$, such that $(X, i, Y, j) \in R_\alpha$ means that α relates the ith element of $\tilde{X}$ (the stream of values exchanged through the port or memory cell X) with the jth element of $\tilde{Y}$ (the stream of values exchanged through Y). The exact meaning of "relates" depends on the nature of the data constraints that appear in the transitions of α. Intuitively, the transitive closure of this relation, R^*_α, gives all symbolic relations that can be derived from the relations in R_α.

Generally, both the number and the lengths of the execution paths in the symbolic execution tree of a constraint automaton can become infinite because of the existence of loops. This means that both R_α and R^*_α can potentially be infinite. In this thesis, we assume that the values in one iteration of a loop do not depend on previous iterations. This assumption holds true in all the connectors that we have encountered. Making this assumption enables us to unfold loops a bounded number of times.

In the regular expression of a constraint automaton, loops are encoded in terms of the form $A*$. So, unfolding loops amounts to expanding these terms.

For our example constraint automaton (see figure 4.1b), we obtained $(\{A\}[m\bullet = a]\{B\}[\bullet m \geq 0, b = \bullet m] + \{A\}[m\bullet = a]\{C\}[\bullet m < 0, c = \bullet m])*$ as its regular expression. Since there is only term of the form $A*$ in this regular expression, there is only one loop in this example. During each iteration of the loop, one of two paths may be taken. The two paths are represented using the union term in the regular expression.

To keep the example simple, let us unfold the loop once. To do this, we would have to expand the regular expression once to obtain the unfolded instance $\{A\}[m\bullet = a]\{B\}[\bullet m \geq 0, b = \bullet m] + \{A\}[m\bullet = a]\{C\}[\bullet m < 0, c = \bullet m]$.

Step 3: Traversing the unfolded instance. In this step, we traverse the unfolded instance of the regular expression of a constraint automaton and compute the data streams of the data variables by indexing the data elements that pass through the port or memory cell. We traverse the terms of the unfolded instance from right to left and specify indices for the each data variable in the data constraints occurring in the regular expression. Then, the transitive closure of the relations among these elements of streams of ports and memory cells can be used to obtain the relation between input and output values.

Step 3.1: Backward indexing. The purpose of indices is to show the order in which data elements appeared at a port or a memory cell during an execution corresponding to the unfolded instance of a regular expression. To compute indices, we traverse the unfolded instance from right to left. If we observe the data variable p corresponding to a port for the first time in a label in the unfolded instance (the rightmost occurrence), we replace it with the indexed name p^ℓ. If we now see another p, we denote it as p^ℓ_{-1}, and so on.

Indexing of memory cells needs a little more care. The typical scenario is that when we start traversing the unfolded instance from right to left, for each memory cell m, we first see $\bullet m$. Since this is the first reference to $\bullet m$ that we have seen, it means that the stream $\tilde{m}$ has been empty so far and this occurrence of $\bullet m$ is reading a value that was written earlier in the execution (but we haven't encountered yet in our traversal). So, in this case, we replace $\bullet m$ with m^ℓ. As we keep traversing, every subsequent occurrence of $\bullet m$ will be replaced with m^ℓ as well until we see $m\bullet$. When we encounter $m\bullet$, we replace it too with m^ℓ because this is the point where this data element that we have been reading up to now is introduced into the stream $\tilde{m}$. As we continue traversing, if we see $\bullet m$ again, we denote it as m^ℓ_{-1} and the corresponding reference to $m\bullet$ by m^ℓ_{-1} as well, and continue this process.

If after observing an occurrence of $m\bullet$ (which, say, we replaced with m^ℓ_{-i}) we see another occurrence of $m\bullet$ without first encountering $\bullet m$, then we replace it with $m^\ell_{-(i-1)}$ to indicate that a new data element is introduced into the data stream $\tilde{m}$. This element, however, is not read. Similarly, if the first occurrence of memory cell m that we encounter during our traversal is $m\bullet$ (instead of $\bullet m$ as in the above discussion), we again replace $m\bullet$ with m^ℓ.

Additionally, if we encounter a union term (i.e., a term whose subterms are combined with the '+' operator), then the indices are propagated separately through each subterm.

We will now illustrate the indexing process with our example constraint automaton (see figure 4.1b). In the previous step, we obtained $\{A\}[m\bullet = a]\{B\}[\bullet m \geq 0, b = \bullet m] + \{A\}[m\bullet =$

$a]\{C\}[\bullet m < 0, c = \bullet m]$ as its unfolded instance. We traverse this unfolded instance from right to left. Since it is a union term, we will process each subterm separately.

We first traverse the subterm $\{A\}[m\bullet = a]\{C\}[\bullet m < 0, c = \bullet m]$. In this term, we first process the data constraint $[\bullet m < 0, c = \bullet m]$. We observe $\bullet m$ in the data constraint and so we replace it with m^ℓ. Also, we will introduce a new index for the data variable c corresponding to the port C. Thus, we will replace c with c^ℓ. Doing this would give us $\{A\}[m\bullet = a]\{C\}[m^\ell < 0, c^\ell = m^\ell]$. Now, we move on to the next data constraint $[m\bullet = a]$. In this data constraint, we encounter $m\bullet$. Since we have already seen $\bullet m$ which we replaced with m^ℓ, we will replace $m\bullet$ as well with m^ℓ. And we will introduce a new index for a and replace it with a^ℓ. After this, we get $\{A\}[m^\ell = a^\ell]\{C\}[m^\ell < 0, c^\ell = m^\ell]$. Thus, we obtain the following set of relations: $\{m^\ell = a^\ell, m^\ell < 0, c^\ell = m^\ell\}$. Similarly, we can traverse the subterm $\{A\}[m\bullet = a]\{B\}[\bullet m \geq 0, b = \bullet m]$ and obtain another set of relations: $\{m^\ell = a^\ell, m^\ell \geq 0, b^\ell = m^\ell\}$.

Step 3.2: Transitive closure. In the final step, we compute the transitive closure of each set of relations obtained in the previous step. Then, in each set, we pick the relations containing only variables corresponding to input and output ports and combine them using the logical and ($\wedge$) operator. Finally, the expression obtained from each set is combined with the logical or ($\vee$) operator to obtain the expression giving the symbolic relation between the inputs and outputs of the connector.

To continue our example, the transitive closure of the set of relations $\{m^\ell = a^\ell, m^\ell < 0, c^\ell = m^\ell\}$ is $\{m^\ell = a^\ell, m^\ell < 0, c^\ell = m^\ell, a^\ell < 0, c^\ell = a^\ell, c^\ell < 0\}$. If we pick out the relations only containing variables corresponding to input and output ports from this set and combine them using the $\wedge$ operator, we obtain $a^\ell < 0 \wedge c^\ell = a^\ell \wedge c^\ell < 0$. Similarly, the transitive closure of the set of relations $\{m^\ell = a^\ell, m^\ell \geq 0, b^\ell = m^\ell\}$ is $\{m^\ell = a^\ell, m^\ell \geq 0, b^\ell = m^\ell, a^\ell \geq 0, b^\ell = a^\ell, b^\ell \geq 0\}$. We can pick the relations only containing variables corresponding to input and output ports from this set as well and combine them using the $\wedge$ operator to obtain $a^\ell \geq 0 \wedge b^\ell = a^\ell \wedge b^\ell \geq 0$. Finally, we can combine the expressions obtained from the two sets of relations and combine them with the $\vee$ operator to obtain $(a^\ell < 0 \wedge c^\ell = a^\ell \wedge c^\ell < 0) \vee (a^\ell \geq 0 \wedge b^\ell = a^\ell \wedge b^\ell \geq 0)$. This expression gives the symbolic relation between the inputs and outputs of the connector in figure 4.1a.

4.2.3 Checking Data Relay Compatibility

We can now define data relay compatibility between connectors on the basis of their respective relations between their inputs and outputs which can be obtained by symbolic execution of their constraint automata. We introduce two notions of compatibility: strong compatibility and weak compatibility.

Definition 9 (Port mapping)**.** *Let* $\alpha_1 = (Q_1, (P_1^{all}, P_1^{in}, P_1^{out}), M_1, \longrightarrow_1, Q_1^0)$ *and* $\alpha_2 = (Q_2, (P_2^{all}, P_2^{in}, P_2^{out}), M_2, \longrightarrow_2, Q_2^0)$ *be two constraint automata. A function* $\mathcal{M} : P_1^{in} \cup P_1^{out} \rightarrow P_2^{in} \cup P_2^{out}$ *is a port mapping if it satisfies the following conditions:*

1. *$\mathcal{M}$ is a total function. In other words, $\mathcal{M}$ maps every input and output port of A_1 to some port of A_2.*
2. *$\mathcal{M}$ is a one-to-one function. In other words, each input and output port of A_1 is mapped to a unique port of A_2.*

3. *$\mathcal{M}$ maps input ports of A_1 to input ports of A_2 and output ports of A_1 to output ports of A_2. Formally, if $p \in P_1^{in}$, then $\mathcal{M}(p) \in P_2^{in}$. Similarly, if $p \in P_1^{out}$, then $\mathcal{M}(p) \in P_2^{out}$.*
Note that ensuring this condition does not add any additional burden on the architect as they would have to anyway consider the mapping between input and output ports to replace the connector.

Definition 10 (Expression lifting)**.** *Let $\alpha_1 = (Q_1, (P_1^{all}, P_1^{in}, P_1^{out}), M_1, \longrightarrow_1, Q_1^0)$ and $\alpha_2 = (Q_2, (P_2^{all}, P_2^{in}, P_2^{out}), M_2, \longrightarrow_2, Q_2^0)$ be two constraint automata. A port mapping $\mathcal{M} : P_1^{in} \cup P_1^{out} \rightarrow P_2^{in} \cup P_2^{out}$ defines a lifting operator on expressions involving only variables corresponding to the input and output ports in α_1. Specifically, if ϕ is an expression that involves only the variables for the input and output ports in α_1, then $\mathcal{M}(\phi)$ is the expression obtained by replacing each p_i^ℓ in ϕ, where $p \in P_1^{in} \cup P_1^{out}$, with $\mathcal{M}(p)_i^\ell$.*

Definition 11 (Strong compatibility)**.** *Let $\alpha_1 = (Q_1, (P_1^{all}, P_1^{in}, P_1^{out}), M_1, \longrightarrow_1, Q_1^0)$ and $\alpha_2 = (Q_2, (P_2^{all}, P_2^{in}, P_2^{out}), M_2, \longrightarrow_2, Q_2^0)$ be the constraint automata corresponding to two connectors. Let ϕ_1 and ϕ_2 be the expressions obtained from the symbolic execution of α_1 and α_2, respectively. Let $\mathcal{M} : P_1^{in} \cup P_1^{out} \rightarrow P_2^{in} \cup P_2^{out}$ be a port mapping. Then the two connectors are said to be strongly compatible with each other with respect to data relay behavior if and only if $\mathcal{M}(\phi_1) \Leftrightarrow \phi_2$.*

Definition 12 (Weak compatibility)**.** *Let $\alpha_1 = (Q_1, (P_1^{all}, P_1^{in}, P_1^{out}), M_1, \longrightarrow_1, Q_1^0)$ and $\alpha_2 = (Q_2, (P_2^{all}, P_2^{in}, P_2^{out}), M_2, \longrightarrow_2, Q_2^0)$ be the constraint automata corresponding to two connectors. Let ϕ_1 and ϕ_2 be the expressions obtained from the symbolic execution of α_1 and α_2, respectively. Let $\mathcal{M} : P_1^{in} \cup P_1^{out} \rightarrow P_2^{in} \cup P_2^{out}$ be a port mapping. Then the two connectors are said to be weakly compatible with each other with respect to data relay behavior if and only if $\mathcal{M}(\phi_1) \Rightarrow \phi_2$.*

4.2.4 Composition of Constraint Automata

So far in this thesis, our discussion has involved connector evolution scenarios in which a single connector is replaced with a single connector. But there may be situations when multiple connectors in a given software system are replaced with a single connector or a single connector is replaced with multiple connectors. As an example, consider the system shown in figure 4.2. As can be seen in the figure, there are three components in the system. Component 1 and component 2 interact using a pipe connector, and component 1 and component 3 have another pipe connector between them. Let's suppose that in this system, component 1 generates a series of data values that have to be sent to component 2 and component 3. Every time a new data value is generated by component 1, it is sent simultaneously to both component 2 and component 3 via the pipe connector by which they are respectively attached to component 1.

The constraint automata for the two pipe connectors in the system whose architecture is shown in figure 4.2 are given in figure 4.3. Figure 4.3a shows the constraint automaton for the pipe connector that is attached to port A of component 1 and port C of component 2. The automaton has two states – state 0 and state 1. State 0 is the start state. In state 0, the connector can accept an incoming data value on port A and it transitions to state 1 after storing the incoming data value in the memory cell $m1$. Then the connector writes the value in the memory cell $m1$ to port C and transitions to state 0. The constraint automaton for the pipe connector that is attached to port B of component 1 and port D of component 3 is shown in figure 4.3b. The behavior of this constraint

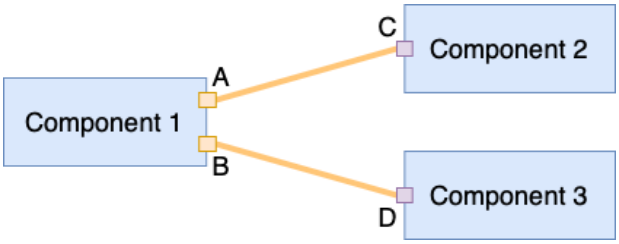
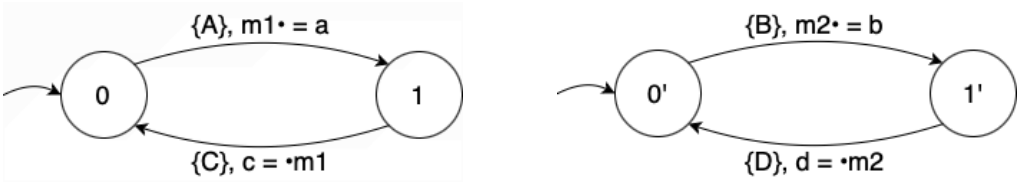


Figure 4.2: A simple system with two pipe connectors that need to be replaced



(a) (b)

Figure 4.3: Constraint automata for the two pipe connectors in the system whose architecture is shown in figure 4.2

automaton is similar to that of the automaton in figure 4.3a which has been explained above.

Suppose that the two pipe connectors in our example system have to be replaced by a single connector. Moreover, suppose that since the same data is being sent through ports A and B, the architects decide to merge the two ports into a single port and use that merged port with the replacement connector. In this case, to check data relay compatibility between the two connectors to be replaced and the replacement connector, we need to combine the constraint automata for the two pipe connectors into a single automaton that describes the overall behavior of the two connectors. In this section, we describe how that can be done.

It might appear that the combined constraint automaton may be defined in a similar way as the parallel composition of transition systems [19]. However, this doesn't work as desired. The reason is that in our example system, the same data is sent at both ports A and B. Parallel composition, however, doesn't capture this. The reason for this is that parallel composition allows transitions occurring when a data item arrives at port A or port B to remain separate. As a result, it doesn't capture the fact that the data values sent through those two ports are always the same because of which they can be merged and it is sufficient to send the data item just once through the merged port. We will now show how a composition operator can be defined for constraint automata that doesn't have this limitation.

First, we will lay out some preliminary definitions.

Definition 13 (Mergeable ports)**.** *Let* $\alpha_1 = (Q_1, (P_1^{all}, P_1^{in}, P_1^{out}), M_1, \longrightarrow_1, Q_1^0)$ *and* $\alpha_2 = (Q_2, (P_2^{all}, P_2^{in}, P_2^{out}), M_2, \longrightarrow_2, Q_2^0)$ *be two constraint automata. Let* $p_1 \in P_1^{all}$ *and* $p_2 \in P_2^{all}$ *be two ports. Ports* p_1 *and* p_2 *are said to be mergeable if whenever a data item arrives at one of the two ports, a data item arrives at the other port as well and the two data items have the same value. Intuitively, two ports are said to be mergeable if the same data values are sent simultaneously through both of them always.*

We define $\mathbf{mergeable}(\alpha_1, \alpha_2)$ *to be the set of pairs of ports in* α_1 *and* α_2 *that are mergeable.*

In other words, if $(p_1, p_2) \in \mathbf{mergeable}(\alpha_1, \alpha_2)$*, then* $p_1 \in P_1^{all}$*,* $p_2 \in P_2^{all}$*, and* p_1 *and* p_2 *are mergeable ports.*

We also define an operator $\langle\rangle$ *as follows. Let* $P \subseteq P_1^{all}$*. Then* $\langle P \rangle = \{p_1 \in P \mid \exists p_2 \in P_2^{all}.(p_1, p_2) \in \mathbf{mergeable}(\alpha_1, \alpha_2)\}$*. Similarly, if* $P \subseteq P_2^{all}$*, then* $\langle P \rangle = \{p_2 \in P \mid \exists p_1 \in P_1^{all}.(p_1, p_2) \in \mathbf{mergeable}(\alpha_1, \alpha_2)\}$.

Note that the elements of the set $\mathbf{mergeable}(\alpha_1, \alpha_2)$ are pairs of ports and this is justified because we are combining the behaviors of only two connectors and usually, at most one port from a connector appears in a particular component. Note that only ports appearing in the same component can be merged. We will describe later how the constraint automata for more than two connectors can be combined.

Definition 14 (Renaming of mergeable ports)**.** *Let* $\alpha_1 = (Q_1, (P_1^{all}, P_1^{in}, P_1^{out}), M_1, \longrightarrow_1, Q_1^0)$ *and* $\alpha_2 = (Q_2, (P_2^{all}, P_2^{in}, P_2^{out}), M_2, \longrightarrow_2, Q_2^0)$ *be two constraint automata. Suppose* $\mathbf{mergeable}(\alpha_1, \alpha_2)$ *is given. Moreover, let* $\mathcal{PN}$ *be the set of all possible port names. Then* $\mathbf{rename} : P_1^{all} \cup P_2^{all} \rightarrow \mathcal{PN}$ *is a function that assigns a unique common name to each port in a pair of mergeable ports. Formally, if* $(p_1, p_2) \in \mathbf{mergeable}(\alpha_1, \alpha_2)$*, then* $\mathbf{rename}(p_1) = \mathbf{rename}(p_2)$*. Moreover, if* $(p_1, p_2) \in \mathbf{mergeable}(\alpha_1, \alpha_2)$ *and* $p_3 \in P_1^{all} \cup P_2^{all}$ *is a port such that* $p_3 \neq p_1$ *and* $p_3 \neq p_2$*, then* $\mathbf{rename}(p_1) \neq \mathbf{rename}(p_3)$ *and* $\mathbf{rename}(p_2) \neq \mathbf{rename}(p_3)$*. The function* $\mathbf{rename}$ *leaves unchanged the names of ports that are not merged.*

Definition 15 (Lifting of port renaming function)**.** *Let* $\alpha_1 = (Q_1, (P_1^{all}, P_1^{in}, P_1^{out}), M_1, \longrightarrow_1, Q_1^0)$ *and* $\alpha_2 = (Q_2, (P_2^{all}, P_2^{in}, P_2^{out}), M_2, \longrightarrow_2, Q_2^0)$ *be two constraint automata. Suppose* $\mathbf{mergeable}(\alpha_1, \alpha_2)$ *is given. Suppose that the function* $\mathbf{rename} : P_1^{all} \cup P_2^{all} \rightarrow \mathcal{PN}$ *is given as well. The function* $\mathbf{rename}$ *defines a lifting operator on sets of ports as well as data constraints. Specifically, if* $P \subseteq P_1^{all} \cup P_2^{all}$*, then* $\mathbf{rename}(P) = \{\mathbf{rename}(p) \mid p \in P\}$*. Additionally, if* ϕ *is a data constraint, then* $\mathbf{rename}(\phi)$ *is the data constraint obtained by replacing each data variable* p *corresponding to a port with* $\mathbf{rename}(p)$.

Now, we're ready to define how the composition of two constraint automata should be constructed.

Definition 16 (Composite constraint automaton)**.** *Let* $\alpha_1 = (Q_1, (P_1^{all}, P_1^{in}, P_1^{out}), M_1, \longrightarrow_1, Q_1^0)$ *and* $\alpha_2 = (Q_2, (P_2^{all}, P_2^{in}, P_2^{out}), M_2, \longrightarrow_2, Q_2^0)$ *be two constraint automata. Suppose* $\mathbf{mergeable}(\alpha_1, \alpha_2)$ *is given. Suppose that the function* $\mathbf{rename} : P_1^{all} \cup P_2^{all} \rightarrow \mathcal{PN}$ *is given as well. The composite constraint automaton of* α_1 *and* α_2*, denoted by* $\alpha_1 \odot \alpha_2$*, is the constraint automaton* $(Q_1 \times Q_2, (\mathbf{rename}(P_1^{all} \cup P_2^{all}), \mathbf{rename}(P_1^{in} \cup P_2^{in}), \mathbf{rename}(P_1^{out} \cup P_2^{out})), M_1 \cup M_2, \longrightarrow, Q_1^0 \times Q_2^0)$ *where* $\longrightarrow$ *is defined by the following rules:*

$$\frac{\begin{array}{c} q \xrightarrow{P_1, \phi_1}_1 q' \\ q'' \xrightarrow{P_2, \phi_2}_2 q''' \\ \langle P_1 \rangle \neq \varnothing \\ \langle P_2 \rangle \neq \varnothing \\ P_1 \setminus \langle P_1 \rangle = \varnothing \\ P_2 \setminus \langle P_2 \rangle = \varnothing \\ \mathbf{rename}(\langle P_1 \rangle) = \mathbf{rename}(\langle P_2 \rangle) \end{array}}{(q, q'') \xrightarrow{\mathbf{rename}(P_1 \cup P_2), \mathbf{rename}(\phi_1 \wedge \phi_2)} (q', q''')}$$

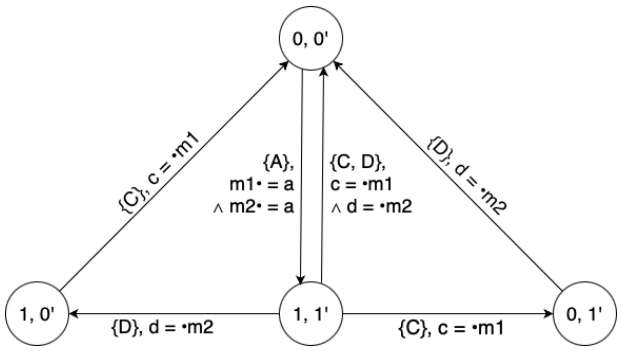


Figure 4.4: Composite constraint automaton for the two constraint automata shown in figure 4.3

$$\frac{q \xrightarrow{P_1,\phi_1}_1 q' \quad \langle P_1 \rangle = \varnothing \quad q'' \xrightarrow{P_2,\phi_2}_2, q''' \quad \langle P_2 \rangle = \varnothing}{(q, q'') \xrightarrow{P_1 \cup P_2, \phi_1 \wedge \phi_2} (q', q''')}$$

$$\frac{q \xrightarrow{P_1,\phi_1}_1 q' \quad \langle P_1 \rangle = \varnothing \quad q'' \in Q_2}{(q, q'') \xrightarrow{P_1,\phi_2} (q', q'')}$$

$$\frac{q \in Q_1 \quad q' \xrightarrow{P_2,\phi_2}_2 q'' \quad \langle P_2 \rangle = \varnothing}{(q, q') \xrightarrow{P_2,\phi_2} (q, q'')}$$

Intuitively, the above rules can be understood as follows. Suppose the constraint automaton α_1 is in state q and the constraint automaton α_2 is in state q''. Also, suppose that the state transition $q \xrightarrow{P_1,\phi_1}_1 q'$ can occur in α_1 and the state transition $q'' \xrightarrow{P_2,\phi_2}_2 q'''$ can occur in α_2. Now, suppose that P_1 and P_2 only include ports that are to be merged. Suppose that P_1 contains the ports in α_1 that each of the ports in P_2 are to be merged with and P_2 contains the ports in α_2 that each of the ports in P_1 are to be merged with. In this case, the two transitions occur synchronously in the composite automaton. They are precluded from occurring independently of each other. This is because the sequence of data values sent through a pair of merged ports is exactly the same and so they can be replaced with a single port of a replacement connector. This is captured in the first rule above. The remaining rules capture the fact that all other pairs of transitions may occur synchronously or independently.

Note that if $\mathbf{mergeable}(\alpha_1, \alpha_2) = \varnothing$, then the composite constraint automaton, $\alpha_1 \odot \alpha_2$, constructed using the above rules would be the parallel composition of α_1 and α_2.

In our example system (see figure 4.2 and figure 4.3), let α_1 be the constraint automaton shown in figure 4.3a and α_2 be the constraint automaton shown in figure 4.3b. In this case, $\mathbf{mergeable}(\alpha_1, \alpha_2) = \{(A, B)\}$. The **rename** function can be defined as $\{A \mapsto A, B \mapsto A, C \mapsto C, D \mapsto D\}$. Figure 4.4 shows the composite automaton, $\alpha_1 \odot \alpha_2$, that would be produced using the above rules.

So far, we have described how the composite constraint automaton for a pair of constraint automata can be constructed. If more than two constraint automata need to be combined, then the composite constraint automaton is constructed in an iterative manner. First, the composite

constraint automaton of the first two connectors is constructed. Next, the composite constraint automaton of the result and the third connector is constructed. This process is continued until the final composite constraint automaton is obtained.

4.3 Implementation

We have implemented the connector compatibility analysis approach laid out above in our programming language WYVERN. We provide a mechanism for specifying the data transfer semantics of a connector. We have implemented a tool that checks whether the given connectors are compatible using the symbolic execution algorithm described in section 4.2.2. We have also provided support for constructing composite constraint automata during compatibility checking in our tool. This allows compatibility checking in connector evolution scenarios where multiple connectors are replaced with a single connector or vice versa.

We have provided a domain-specific language (DSL) for specifying the constraint automaton for a connector. Figure 4.5 shows how the constraint automaton shown in figure 4.1b can be specified using our DSL. The constraint automaton specification is started by using the `automaton` keyword as can be seen in line 1 of figure 4.5. Line 2 specifies that the constraint automaton has two states – `q1` and `q2`. The start state of the automaton – `q1` – is given in line 3. Line 4 specifies that the constraint automaton has a single memory cell named `m`. The transitions of the constraint automaton are specified in lines 5 to 8. For each transition, the start state and the end state are given. This is following by specifying the ports involved in the transition as well as the data constraint that must hold for the transition to occur. For example, the transition specified on line 6 is from state `q1` to state `q2`. Only one port is involved in this transition – port `A`. The data constraint for this transition is `m dot = data A`. Here, `m dot` stands for $m\bullet$ and `data A` stands for the data variable a which is the data variable corresponding to port A. The other two transitions on lines 7 and 8 can be interpreted in a similar manner.

In WYVERN, the constraint automaton specification of a connector is embedded in the metadata of the type corresponding to the connector. (For details on connector type metadata, see section 3.3.2. WYVERN provides a mechanism for implementing DSLs that is called type-specific languages (TSLs) [82]. A TSL may be thought of as a DSL that specifies how a value of a particular type can be constructed. A type in WYVERN can have a TSL associated with it. Values of this type can then be specified using the TSL instead of using an API or other primitives of the language. For specifying the constraint automaton for a connector, we introduce an optional field named `data_semantics` in the metadata for the type corresponding to the connector. This field has the type `ConstraintAutomaton`. We implement our DSL as a TSL associated with the `ConstraintAutomaton` type. Figure 4.6 shows how the constraint automaton for our example connector is specified in the type for that connector which is named `BranchingConnector` (line 1). The definition of the `data_semantics` field can be seen in lines 5 to 13. The `~` after the equal to sign on line 5 is used to indicate that the value of the field should be constructed using the TSL specification that follows starting in the next line.

We have implemented a tool that analyzes whether a specified replacement connector is compatible with a connector that is to be replaced. The tool implements the symbolic execution algorithm explained in section 4.2.2. To run the compatibility analysis tool, the type for the

```
automaton
    states: q1, q2
    start: q1
    memory: m
    transitions:
        q1 -> q2 : {A}; m dot = data A
        q2 -> q1 : {B}; dot m >= 0 and data B = dot m
        q2 -> q1 : {C}; dot m < 0 and data C = dot m
```

Figure 4.5: Constraint automaton specification using the DSL in WYVERN

```
type BranchingConnector
    ...
    metadata new
        ...
        val data_semantics: ConstraintAutomaton = ~
            automaton
                states: q1, q2, q3
                start: q1
                memory: m
                transitions:
                    q1 -> q2 : {A}; m dot = data A
                    q2 -> q1 : {B}; dot m >= 0 and data B = dot m
                    q2 -> q1 : {C}; dot m < 0 and data C = dot m
```

Figure 4.6: Constraint automaton specification in connector type metadata

connector to be replaced as well as that for the replacement connector has to be provided to the tool. A port mapping specification that describes which port of the replacement connector maps to which port of the connector to be replaced has to be supplied as well. If multiple connectors are being replaced with a single connector or vice versa, then the ports which can be merged have to specified as well. In this case, the composite constraint automaton is constructed and compatibility analysis is performed using the composite automaton.

Chapter 5

Evaluation

In this chapter, we will validate the claims of our thesis statement that was given in section 1.1 and is restated below:

> *We can make the evolution of connectors in a software system easier by integrating the architecture description of the system with its implementation when building the system. The integration of the architecture description of a system with its implementation can be achieved in a general way so that a wide variety of connectors as well as connector evolution scenarios may be supported. Furthermore, this integration facilitates analysis of the semantic compatibility of different connectors which can aid in the evolution of connectors by enabling architects to assess whether a candidate replacement connector preserves a behavior of interest and thus prevent unintended introduction of errors.*

Our thesis statement is composed of three claims:

1. **Ease of Connector Evolution Via Architecture-Code Integration.** The first claim is that the evolution of connectors in a software system can be made easier by integrating the architecture description of the system with its implementation when building the system. In chapter 3, we presented our approach for integrating the architecture specification of a system with its code. In our approach, we provide a mechanism for explicitly specifying the architecture of the system. The architecture specification describes the topology of the component connections and the type of connector used for each connection. Interfaces for components must also be specified in the architecture description. Components are then implemented such that they interact exclusively via these interfaces. In the component code, we prevent the direct use of libraries that can be used to implement connectors. As explained in chapter 3, we do this by the use of capabilities. By doing this, we prevent developers from bypassing the use of interfaces for component interaction. We provide an explicit abstraction mechanism for connectors. Based on the specification of component interfaces and component-connector attachments in the architecture description, the code that enables components to interact using a particular connector type can be generated using that connector's abstraction. Since components interact only through interfaces and the code implementing a connector is generated using its abstraction, references to a particular connector type are not spread across the codebase as happens in the current state of practice.

Instead, they are localized to the architecture description. As a result, changing a connector becomes easy as only a few lines in the architecture specification have to be changed.

2. **Generality of the Connector Abstraction Mechanism and the Connector Evolution Approach.** The connector abstraction mechanism that we have developed is expressive enough to implement the connectors in use in real-world systems today. Furthermore, the connector evolution approach that we have developed can be used to change connectors in a wide variety of systems that are implemented using these connectors.
3. **Usefulness of Connector Compatibility Analysis in Prevention of Errors.** In current practice, architecturally relevant information about connectors is often included in code. For example, in systems implemented using the Robot Operating System 2 (ROS 2) framework, the message delivery policy can be configured to be either best effort or guaranteed delivery [7]. This is done in the code using an API function call. Previous studies have found that such architectural knowledge can be lost over time [45]. The connector abstraction mechanism we have developed can be used to specify relevant semantics of connectors explicitly rather than leaving it in code. In chapter 4, we discussed how the data transfer semantics of a connector can be specified using constraint automata. This enables checking whether a connector being replaced and the replacement connector are compatible with respect to the specified semantics. By doing this early in the connector migration process, errors can be prevented from being introduced into the system.

In the following sections, we describe how we validated the above three claims.

5.1 Claim 1: Ease of Connector Evolution Via Architecture-Code Integration

We evaluated the claim that integrating the architecture specification of a system with code makes connector evolution easy on two case study scenarios. The two scenarios were selected to be typical of the kinds of connector evolution scenarios that developers are interested in today. In the first case study, we migrate a suite of robotic software systems implemented using the first generation of the Robot Operating System (ROS) framework, i.e., ROS 1, to the second generation, i.e., ROS 2. In the second case study, we migrate a web application that uses a SQL-based database to use a NoSQL database instead. We describe how we evaluated our approach in these two scenarios in the subsections below.

5.1.1 Case Study 1: ROS 1 to ROS 2 Migration

For our first case study, we have a suite of robotic software systems implemented using ROS 1 and we migrated them to use ROS 2 instead.

Overall Setting

The Robot Operating System (ROS) [88] is an open source framework for building robotic software. It provides a collection of libraries and tools for creating software for a wide variety of

robotic platforms.

ROS comes in two flavors - ROS 1 and ROS 2. Since its introduction in 2007, ROS 1 has been widely adopted in various robotics applications [38]. It supports reuse of drivers and algorithms, and provides interoperability with other robotics frameworks such as Orocos. However, ROS 1 lacks support for real-time systems and has significant security issues [41, 69]. This has motivated the creation of a new generation of the framework: ROS 2.

ROS 1- and ROS 2-based systems are structured as independent components called *nodes*, which communicate with each other using one or more of three communication primitives that can be viewed as architectural connectors: *topics*, *services* and *actions*.

Topics are an implementation of a publish/subscribe communication mechanism which allows nodes to send messages to each other asynchronously. ROS allows multiple nodes to publish to a topic. Similarly, multiple nodes may subscribe to a topic and receive messages published to that topic. Whenever a node publishes a message to a topic, the message is delivered to all nodes that have subscribed to that topic. Topics are the most common communication mechanism used among nodes in ROS-based systems. They are used for broadcasting information that frequently gets updated (such as sensor data) as well as for sending commands (such as for turning an actuator on or off).

Services are an implementation of a synchronous remote procedure call mechanism. To avoid blocking a service caller for an extended period of time, they are generally used for performing tasks that require a short time to complete. Examples of such tasks include querying the state of a node and performing a quick inverse kinematics computation.

Actions are an implementation of an asynchronous call/return mechanism. They are used for performing long-running tasks, such as commanding a robot to navigate to a particular location. To initiate the execution of an action, a node sends a message specifying the *goal* to the node that implements the action. After the execution of the action is completed, the *result* is sent to the caller. During execution, *feedback* may be sent to the caller to inform it about the progress toward the goal. The caller may also *cancel* the action if the task takes too long to complete.

As the final official release of ROS 1 has been announced and ROS 1 will no longer be officially supported [6], there is a need to migrate existing robotic software based on ROS 1 to ROS 2. The communication primitives that we described above are exposed to application programmers using different APIs in ROS 1 and ROS 2. So, to migrate the source code of a system from ROS 1 to ROS 2, the communication primitives in the system have to be ported to use ROS 2 APIs. For the first case study, we performed this connector evolution task.

Software Systems Used In the Study

We searched GitHub and selected five systems implemented using ROS 1 to migrate them to ROS 2 in this study. The five systems that we selected are:

1. TurtleBot3 Teleoperation
2. PX4 Control
3. BB8 Square Motion
4. Parrot Drone Square Motion
5. Robotiq Gripper Control

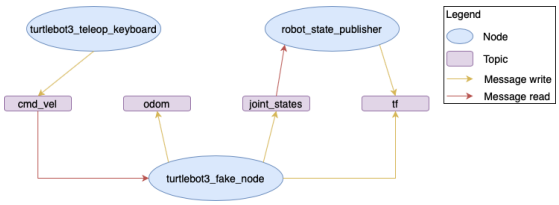


Figure 5.1: Architecture of the TurtleBot3 teleoperation system

We selected these five systems for three reasons:

1. **Manageable size:** Since the systems had to be migrated manually in our study, we required the systems to be of manageable size to complete the study within a reasonable time frame.
2. **Coverage of ROS communication primitives:** We required all three ROS communication primitives (topics, services and actions) to be used in the system(s) that we selected for our study. Since we couldn't find a single system that was of manageable size and also included all three primitives, we selected a suite of five systems instead such that all three communication primitives are present in the five systems taken together.
3. **Ability to run in simulation:** We selected systems that could be run in simulation so that they could be easily tested.

The details of the five systems we used in our case study are as follows:

TurtleBot3 Teleoperation: The TurtleBot3 teleoperation system enables the TurtleBot3[1] robot to be moved remotely by sending it commands from a keyboard. As shown in figure 5.1, the TurtleBot3 teleoperation system consists of three nodes: 1) `turtlebot3_fake_node`, 2) `robot_state_publisher`, and 3) `turtlebot3_teleop_keyboard`. The nodes communicate with each other via topics. The `turtlebot3_teleop_keyboard` node processes keyboard input and converts it to commands for moving the robot which are sent to the `turtlebot3_fake_node` node by publishing them on the `cmd_vel` topic. The `turtlebot3_fake_node` node simulates the motion of the robot in accordance to the teleoperation commands that it reads from the `cmd_vel` topic. It also publishes the amount each of the robot's two controllable wheels have turned as well as their angular velocities on the `joint_states` topic. Additionally, it computes odometry information i.e., an estimate of the position, orientation and velocity of the robot and publishes it on the `odom` topic. Lastly, the `turtlebot3_fake_node` node computes the coordinate frame transform for the robot's base with respect to the estimated odometry information and publishes it on the `tf` topic. The `robot_state_publisher` node computes the forward kinematics of the robot i.e., it computes the positions of the robot's base and all of its joints, and publishes the computed state of the robot on the `tf` topic.

PX4 Control: The PX4 Control system enables the Iris+ drone, developed by 3D Robotics, to be controlled using the PX4 flight control software. The system implements four nodes, each of which is named `test_node`. The four nodes are meant to be run independently of each other. Here, we use subscripts to distinguish between them.

[1]https://www.turtlebot.com

The first two nodes, test_node$_1$ and test_node$_2$, are used to control a single drone, while the other two nodes, test_node$_3$ and test_node$_4$, are used to control a group of three drones. The test_node$_1$ node enables a single drone to be controlled in *offboard* mode, while the test_node$_2$ node enables the drone to be controlled in *mission* mode. In the offboard mode, commands for the drone are input by the user from a keyboard. Based on the command entered by the user, a target velocity for the drone is computed which is published to the mavros/setpoint_raw/local topic. In the mission mode, a flight plan, consisting of a sequence of points, is provided by the user in a file and the points specified in the flight plan are sent to the drone by calling a service named mavros/mission/push. Similarly, the test_node$_3$ node enables a group of three drones to be controlled in offboard mode and the test_node$_4$ enables the group to be controlled in mission mode. In the offboard mode, the user successively selects the specific drone in the group they want to control using the keyboard and then enters the commands to be sent to that drone. In the mission mode, a flight plan is provided by the user which specifies the sequence of points that each drone should travel to.

In each of the four nodes, the service mavros/set_mode is called to put the drone(s) in offboard mode or mission mode, and the service mavros/cmd/arming service is used to arm the drone, i.e., to turn on the propeller motors. Each node also subscribes to the mavros/state topic to receive information about the state of the drone(s).

BB8 Square Motion: The BB8 Square Motion system can be used to make the BB8 robot move in a square path. The system implements four nodes. The move_square_node node implements a service named move_bb8_square which can be called to make the BB8 robot move in a square with a predetermined side length of 2 m. The move_bb8_square service is called from the bb8_activate_client node. Similarly, the move_square_custom_node node implements a service named move_bb8_in_square_custom which is called to make the BB8 robot move in a square with a custom side length specified by the user. The move_bb8_in_square service is called from the call_move_square_custom_node node. To make the BB8 robot move, the move_bb8_square and move_bb8_in_square_custom services publish commands to change the velocity and the travel direction of the robot on the cmd_vel topic.

Parrot Drone Square Motion: The Parrot Drone Square Motion system is similar to the BB8 Square Motion system. It can be used to make a Parrot drone move in a square with a side length that's specified by the user. The system implements two nodes. The parrot_square_node node implements an action named parrot_moving_square_action_server which makes the drone move in a square. This action is invoked from the parrot_moving_square_client_action_node node. In the parrot_moving_square_action_server action, the drone is moved by publishing commands to change its velocity and direction of travel on the cmd_vel topic.

Robotiq Gripper Control: The Robotiq Gripper Control system can be used to operate the two finger grippers 2F-85 and 2F-140 made by Robotiq. The system implements two nodes. The robotiq_2f_action_server node implements an action called command_robotiq_

Table 5.1: ROS communication primitives included in the systems used in the ROS 1 to ROS 2 migration case study

System	Topics	Services	Actions
TurtleBot3 Teleoperation	✓	–	–
PX4 Control	✓	✓	–
BB8 Square Motion	✓	✓	–
Parrot Drone Square Motion	✓	–	✓
Robotiq Gripper Control	✓	–	✓

`action` which can be used to move the gripper fingers to the desired position. This action is invoked from the `robotiq_2f_client` node. During the execution of the action, updates about the state of the joints of the gripper are published to the `joint_states` topic.

Table 5.1 shows the ROS communication primitives included in each of the five systems described above.

Methodology

We compared the number of lines that would need to be changed if the ROS 1 to ROS 2 migration was done using our approach with the number of lines that would have to be changed if the migration were done on the implementation of the systems as it exists. We also compared the distribution of the lines that need to be modified across files when the migration is done with and without using our approach.

To perform the migration using our approach, we first adapted the ROS 1 version of each of the five systems to use our architectural approach.[2] The nodes in the five systems are implemented using C++ or Python. For pragmatic reasons, we left the Python or C++ algorithmic code in each node unchanged, but wrapped the node in a WYVERN module using the foreign function interface of the language. Direct invocations of ROS 1 communication libraries were replaced with (simpler) port interfaces in WYVERN. ROS 1 also provides non-communication libraries; we wrote wrappers for this functionality, as adapting to changes in arbitrary libraries is not in scope for our approach. We then wrote an architecture specification in our ADL. Finally, we implemented connector abstractions for the three ROS communication primitives–topics, services and actions. Note that implementation of the library wrappers as well as connector abstractions is a one-time effort akin to implementation of the ROS 1 middleware and can be reused for subsequent implementations of other systems.

To migrate the systems to ROS 2, we implemented wrappers for ROS 2 libraries providing non-communication-related functionality. We also implemented ROS 2 versions of the connector abstractions. Again, this is a one-time effort analogous to a reusable implementation of the ROS 2 middleware. Once the implementation of the library wrappers and the connector abstractions is done, the connectors can be migrated to ROS 2 by just modifying the architecture specification.

[2]We do not consider this adaptation as part of the cost of our approach, since if our approach is adopted, systems will be implemented this way from the beginning.

Table 5.2: Changes made to migrate connectors in the TurtleBot3 Teleoperation system from ROS 1 to ROS 2

Node	ROS 1 Implementation			ROS 2 Implementation		
	File	# lines	# lines changed	File	# lines	# lines added
turtlebot3_fake_node	turtlebot3_fake.h	115	17	turtlebot3_fake_node.hpp	81	–
	turtlebot3_fake.cpp	258	10	turtlebot3_fake_node.cpp	267	1
robot_state_publisher	joint_state_listener.h	85	8	joint_state_listener.h	90	1
	robot_state_publisher.h	92	3	robot_state_publisher.h	106	–
	joint_state_listener.cpp	201	11	joint_state_listener.cpp	233	–
	robot_state_publisher.cpp	131	8	robot_state_publisher.cpp	181	–
turtlebot3_teleop_keyboard	turtlebot3_teleop_key.py	193	3	teleop_keyboard.py	218	1
Total		1075	60		1176	3

In particular, the ROS 1 connector types must be replaced by their ROS 2 analogues in connector type declarations and connector instance declarations. To complete the migration, the uses of ROS 1 library wrappers must be changed to equivalent ROS 2 ones which is a manual process. Alternatively, a higher level abstraction may be defined for these libraries and implemented for both ROS 1 and ROS 2. Migrating the systems then would involve switching to the appropriate implementation of the library abstractions.

For comparison, we need the ROS 2 versions of the systems created without using our approach where the changes have been made in the ROS 1 implementation of those systems as it exists. For the TurtleBot3 Teleoperation system, the developers of the system have already migrated it to ROS 2. So, we used that. That also provides us with a benchmark for how ROS experts migrate a system to ROS 2. For the remaining four systems, we carried out the migration to ROS 2 ourselves. This involved replacing calls to ROS 1 library functions–for both communication primitives and non-communication functionality–with their ROS 2 counterparts. To compare with our approach, we only count changes made to lines that implement communication operations.

Results

We will now analyze the effectiveness of our approach in making the task of connector evolution easier when migrating systems from ROS 1 to ROS 2. Below we present the results for each of the five systems we have used in our case study.

TurtleBot3 Teleoperation: As mentioned above, the TurtleBot3 Teleoperation system consists of three nodes which are named `turtlebot3_fake_node`, `robot_state_publisher` and `turtlebot3_teleop_keyboard`. The `turtlebot3_fake_node` and `robot_state_publisher` nodes have been implemented in C++, whereas the `turtlebot3_teleop_keyboard` node has been implemented in Python. The source files implementing these nodes in ROS 1 and ROS 2 with their respective number of lines (including comments) as well as the number of lines in the ROS 1 version that had to be changed for changing the connectors to ROS 2 are shown in table 5.2. Note that the names of some files have been changed between ROS 1 and ROS 2 (e.g., from turtlebot3_fake.h to turtlebot3_fake_node.hpp).

As we noted above, the ROS 2 version of the TurtleBot3 Teleoperation system was created by expert developers. Four kinds of changes have been made to carry out the migration to ROS 2:

1) A line in the ROS 1 code is removed (e.g., removal of `using` directives involving ROS 1 namespaces) [11 lines]. 2) A line in the ROS 1 code is replaced by a single line (e.g., using the appropriate ROS 2 function for publishing/subscribing to a topic) [46 lines]. 3) A line in the ROS 1 code is replaced by more than one line (e.g., publishing/subscribing to a topic while also specifying a queue size) [3 lines]. 4) A line is added to the ROS 2 code (e.g., declaration of a `Node` variable) [3 lines]. Thus, a total of 60 lines has been modified in the ROS 1 implementation and 3 new lines have been added to the ROS 2 implementation for migrating the connectors.

In comparison, migration of connectors in our approach only involves changing the architecture specification. The architecture specification of the TurtleBot3 teleoperation system consists of 39 lines. Changing ROS 1 connectors to ROS 2 can be accomplished by changing just 4 lines in the architecture specification, which is substantially easier to do than carrying out the changes in the original implementation by experts. These changes involve modifying the connector type declarations and then changing the connector instance declarations to use the appropriate connector types.

PX4 Control: Table 5.3 shows the nodes in the PX4 Control system as well as the source code files that implement them. It also shows the number of lines (including comments) in these files in both the ROS 1 and ROS 2 versions of the system. The number of lines that had to be modified in each file to migrate the system to ROS 2 is also shown. The modifications made to the ROS 1 version of the system falls into one of three categories: 1) replacement of a line with a single line [98 lines], 2) replacement of a line with multiple lines [7 lines], and 3) addition of new lines [4 lines]. The addition of new lines corresponds to extra steps that need to be performed to initialize and use the ROS 2 communication primitives. As can be seen from the table, a total of 105 lines has been modified in the ROS 1 implementation of the PX4 Control system and 4 new lines have been added to the ROS 2 implementation for migrating the ROS-based connectors.

As mentioned previously, the nodes in the PX4 Control system implement four scenarios: 1) operation of a single drone in offboard mode, 2) operation of a single drone in mission mode, 3) operation of a group of three drones in offboard mode, and 4) operation of a group of three drones in mission mode. We created a separate architecture specification file for each of these four scenarios in WYVERN. The architecture specification for scenarios 1 and 2 each consists of 41 lines and that for scenarios 3 and 4 each consists of 45 lines. To migrate the connectors in the system to ROS 2, we had to change just 7 lines in the architecture specification for scenarios 1 and 2 each, and just 8 lines in the architecture specification for scenarios 3 and 4 each. As in the case of the TurtleBot3 Teleoperation system, these changes involve modifying the connector type declarations and the connector instance declarations to specify that ROS 2 versions of the connectors should be used.

BB8 Square Motion: Table 5.4 shows the nodes implemented in the BB8 Square Motion system. It shows the files used to implement these nodes as well. The number of lines (including comments) in the ROS 1 and ROS 2 versions of each file as well as the number of lines in the ROS 1 version that had to be changed to migrate the ROS communication primitives used to ROS 2 are also shown. As in the case of the above systems, the changes made to the ROS 1 version of the system are of three types: 1) replacement of a line with a single line [48 lines], 2)

Table 5.3: Changes made to migrate connectors in the PX4 Control system from ROS 1 to ROS 2

Node	File	ROS 1 Implementation		ROS 2 Implementation	
		# lines	# lines changed	# lines	# lines added
test_node$_1$	manual_px4_ros.py	288	23	324	1
test_node$_2$	px4_ros_mission.py	271	21	293	1
test_node$_3$	multi_manual_px4_ros.py	264	25	300	1
test_node$_4$	multi_uav_px4_ros.py	73	7	81	1
	px4_ros.py	186	29	213	–
Total		1082	105	1211	4

Table 5.4: Changes made to migrate connectors in the BB8 Square Motion system from ROS 1 to ROS 2

Node	File	ROS 1 Implementation		ROS 2 Implementation	
		# lines	# lines changed	# lines	# lines added
move_square_node	bb8_move_in_square_service_server.py	25	8	32	1
	move_bb8.py	70	12	81	–
bb8_activate_client	bb8_move_in_square_service_client.py	46	10	53	1
move_square_custom_node	bb8_move_custom_service_server.py	39	8	48	1
call_move_square_custom_node	bb8_move_custom_service_client.py	48	14	56	1
Total		228	52	270	4

replacement of a line with multiple lines [4 lines], and 3) addition of new lines [4 lines]. So, a total of 52 existing lines had to be changed and 4 new lines (corresponding to an operation that didn't have to be performed in the ROS 1 version) had to be added.

As we described earlier, two scenarios are implemented in the BB8 Square Motion system: 1) moving the BB8 robot in a square with a predefined side length, and 2) moving the robot in a square with a custom side length that's specified by the user. To implement the system in our approach, we created separate architecture specifications for the two scenarios. The architecture specification for each scenario consists of 28 lines. To change the ROS 1 connectors to ROS 2, we had to change just 4 lines in each architecture specification. As in the above systems, these changes correspond to changing the connector type declarations and the connector instance declarations to use the ROS 2 versions of the connectors.

Parrot Drone Square Motion: Table 5.5 shows the nodes that have been implemented in the Parrot Drone Square Motion system as well as the source code files in which they have been implemented. For each file, the number of lines (including comments) in the ROS 1 and ROS 2 versions of the files can be seen in the table. The table also shows the number of lines that had to be changed in the ROS 1 version of the files to change the ROS-based connectors to ROS 2. Again, the changes made in the files to migrate the connectors used from ROS 1 to ROS 2 are of three kinds: 1) replacement of a line with a single line [31 lines], 2) replacement of a line with multiple lines [5 lines], and 3) addition of new lines [2 lines]. Thus, a total of 36 lines in the two files had to be changed and 2 new lines had to be added to change to the ROS 2 versions of the connectors that have been implemented in the system.

In the implementation of the system with our approach, we created an architecture specification

Table 5.5: Changes made to migrate connectors in the Parrot Drone Square Motion system from ROS 1 to ROS 2

Node	File	ROS 1 Implementation		ROS 2 Implementation	
		# lines	# lines changed	# lines	# lines added
parrot_square_node	parrot_moving_square_server_node.py	192	25	208	1
parrot_moving_square_client_action_node	parrot_moving_square_client_node.py	84	11	98	1
Total		276	36	396	2

Table 5.6: Changes made to migrate connectors in the Robotiq Gripper Control system from ROS 1 to ROS 2

Node	File	ROS 1 Implementation		ROS 2 Implementation	
		# lines	# lines changed	# lines	# lines added
robotiq_2f_action_server	robotiq_2f_action_server.py	238	38	261	1
	robotiq_2f_gripper_driver.py	544	18	572	–
robotiq_2f_client	robotiq_2f_action_client_example.py	61	13	74	1
Total		843	69	907	2

for the system. The architecture specification is comprised of 35 lines. To change the connectors in the system from the ROS 1 versions to their ROS 2 counterparts, we had to change just 6 lines in the architecture specification. Similar to the above systems, these changes correspond to changing the connector type declarations and the connector instance declarations to use the ROS 2 versions of the connectors.

Robotiq Gripper Control: Table 5.6 shows the nodes implemented in the Robotiq Gripper Control system along with the source code files that have been used to implement them. The table also shows the number of lines (including comments) in both the ROS 1 and ROS 2 versions of each file. Additionally, the number of lines that have been changed in the ROS 1 version of each file to change the connectors used in the system to ROS 2 is shown as well. Once again, the modifications that have been made to the file to migrate the connectors used from ROS 1 to ROS 2 are of three types: 1) replacement of a line with a single line [58 lines], 2) replacement of a line with multiple lines [11 lines], and 3) addition of new lines [2 lines]. So, a total of 69 existing lines had to be changed and 2 new lines had to be added to change to the ROS 2 versions of the connectors that have been implemented in the system.

For implementing the system using our approach, we created an architecture specification of the system. The architecture specification consists of 29 lines. To change the connectors in the system from the ROS 1 versions to their ROS 2 counterparts, we had to change just 4 lines in the architecture specification. As in the above systems, these changes correspond to changing the connector type declarations and the connector instance declarations to use the ROS 2 versions of the connectors.

Summary: In summary, to change the connectors in the five systems we have used in our case study from ROS 1 to ROS 2, about 50 to 100 lines had to be changed in each system. Moreover, these lines were spread across all the files that were used for implementing the nodes in the system. So, without our approach, the task of making these changes is quite tedious. By contrast, in our approach, we had to change only the architecture specification of each system. And we had to

change only about 10 lines in each architecture specification. This is substantially easier to do than making the modifications in the original implementation that doesn't use our approach.

Threats to Validity

With respect to external validity, we note that we have evaluated our approach on open source software. The connectors in the systems we have used have been implemented in accordance to the guidelines provided by the developers of ROS in the documentation for ROS. Our results may not generalize to proprietary systems that have been implemented differently.

With respect to construct validity, we note that we have used number of lines of code as a measure of the difficulty of performing the connector evolution task. However, there might be other factors that have an influence on the difficulty of the task of connector evolution. It might be possible to uncover those factors only in a real development setting. In other words, our measure of effort might only be an approximation and a true estimate would require observing expert developers using our approach in their projects.

Additional Benefits

In addition to localizing changes to the architecture specification, our architecture-centric connector evolution approach provides additional benefits. As we discussed in chapter 3, our approach ensures *data uniformity* after a connector change has been carried out in a system. In other words, our approach ensures that the data read from or written to a connector's replacement by a component attached to it remains the same as the data it read from or wrote to the original connector. Additionally, our approach ensures *semantic equivalence of individual components* as well. In other words, our approach ensures that there is no change in the behavior of individual components after connector migration has been carried out. These benefits are seen in our ROS 1 to ROS 2 migration case study in the systems we have used in our study.

Data Uniformity: Previous studies have found that refactoring operations can potentially introduce bugs [27]. In the original implementation of ROS-based systems, bugs might potentially be introduced while changing connectors from ROS 1 to ROS 2 which would cause data errors in the ROS 2 version of the system. For example, the ROS 1 implementation of the `turtlebot3_fake_node` node in the TurtleBot3 Teleoperation system uses the following code to create a publisher to the `odom` topic:

```
odom_pub_= nh_.advertise<nav_msgs::Odometry>("odom", 100);
```

In ROS 2, this is done using the following code:

```
auto qos = rclcpp::QoS(rclcpp::KeepLast(100));
odom_pub_= this->create_publisher<nav_msgs::msg::Odometry>("odom", qos);
```

Thus, in order to change to ROS 2, `nh_.advertise` has to be modified to `this->create_publisher`, `nav_msgs::Odometry` to `nav_msgs::msg::Odometry` and the method argument `100` to `qos`. While making these changes, the developer can potentially also change

the string literal argument `"odom"` inadvertently and thus introduce a bug resulting in publishing messages to an incorrect topic. This would result in a violation of data uniformity because a node that subscribes to the `odom` topic would stop receiving messages published by the node in which the bug is present. In their work on detecting such architecture misconfiguration bugs in ROS-based systems, Timperley et al. have found that these kinds of bugs do get introduced into systems during their evolution and are quite difficult to find and fix [102].

Such errors are less likely to occur in our approach because the changes are localized to the architecture specification. The only places that would require modification are connector type declarations and the connector type specification in connector instance declarations. So, developers can consciously avoid changing connector properties. It is also easy to prevent this error in our approach by implementing lint-like checks to ensure that the connector properties are not changed during migration. Doing this in the original implementation would require architecture extraction from code, which can be quite challenging. However, in our approach, the architecture description is explicitly specified and can be statically analyzed to implement such checks.

Semantic Equivalence of Individual Components: There is also potential for introducing semantic anomalies while changing the connectors in the original implementation of ROS-based systems from ROS 1 to ROS 2. For example, in the TurtleBot3 Teleoperation system, to subscribe to the `cmd_vel` topic, the following code is used in the ROS 1 implementation of the `turtlebot3_fake_node` node:

```
cmd_vel_sub_= nh_.subscribe("cmd_vel", 100,
   &Turtlebot3Fake::commandVelocityCallback, this);
```

To change to ROS 2, this is modified as follows:

```
cmd_vel_sub_= this->create_subscription<geometry_msgs::msg::Twist>(
   "cmd_vel", qos, std::bind(
     &Turtlebot3Fake::command_velocity_callback,
     this, std::placeholders::_1));
```

In this case, the template argument `geometry_msgs::msg::Twist` has to be specified explicitly as the compiler is unable to deduce it. This introduces the potentiality to get it wrong, which would result in a compilation error.

Again, the callback function that is used to subscribe to the `cmd_vel` topic has the following signature:

```
void Turtlebot3Fake::commandVelocityCallback(const
   geometry_msgs::TwistConstPtr cmd_vel_msg)
```

To migrate the connectors to ROS 2, the signature of the callback function has to be changed to:

```
void Turtlebot3Fake::command_velocity_callback(const
   geometry_msgs::msg::Twist::SharedPtr cmd_vel_msg)
```

While changing the connector to ROS 2, a developer can potentially leave out the `::SharedPtr` part from the type in the parameter declaration. This would also lead to a compilation error.

As can be seen from the systems we have used in our case study, ROS-based systems are commonly implemented using Python. The kinds of type errors that we have described above can be introduced into systems implemented in Python as well. However, since Python is a dynamic language, these errors would be manifested during run-time in systems implemented in Python and thus would be harder to debug.

These errors cannot occur with our approach because the code for connector implementation is automatically generated.

5.1.2 Case Study 2: SQL to NoSQL Migration

For our second case study, we have a web application that uses MySQL, which is a SQL-based database. and we migrated it to use MongoDB, which is a NoSQL database, instead.

Overall Setting

Traditionally, web applications have been implemented using a relational database for data persistence. Data is stored in a structured format in relational databases and accessed using queries written in the Structured Query Language (SQL).

As the traffic on a web application grows and the volume and velocity of data processed by the application increases, relational databases can become a performance bottleneck. The common tactic of improving performance by horizontal scaling doesn't work well with relational databases. In other words, the performance of relational databases cannot be improved by simply spreading the data across multiple servers. This is because horizontal scaling introduces a large processing overhead to fulfill the ACID (atomicity, consistency, isolation and durability) guarantees provided by relational databases [37].

To overcome the limitations of relational databases, databases that store data in non-relational formats have been developed. Such databases are colloquially referred to as NoSQL databases because many of them don't support SQL queries.

NoSQL databases are generally classified into four categories: 1) key-value stores, 2) document databases, 3) column family databases, and 4) graph databases. In *key-value stores*, each item in the database is stored as a pair consisting of a key and the corresponding value. An item is accessed by providing the key associated with it. *Document databases* store data as semi-structured entities called documents which are typically in a standard format such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON). *Column family databases* store data as rows that are associated with a row key. Each row has a set of columns and columns may be grouped into column families. Not every column is present in each row and, as such, column family databases are designed to be efficient at storing sparse data. *Graph databases* store data that's structured as a graph consisting of nodes and edges.

In this case study, we evaluate our approach on the task of migrating a web application that uses a SQL-based database to a document-oriented NoSQL database. Our approach is applicable here because even though the database is a component from an architecture standpoint, when the database is changed, the mechanism used to interact with the database would also have to be

changed. Since the interaction mechanism would be modeled as a connector in an architecture specification, connector evolution is also involved in the migration of a web application from a SQL-based database to a NoSQL database.

Methodology

We migrated the Stanford Conference and Research Forum (SCARF) web application from a relational SQL-based database to a NoSQL database. We found this web application by searching the research literature on approaches for performing SQL to NoSQL migration. SCARF has previously been used in other studies [10, 14]. We selected SCARF for our study mainly due to its manageable size. Since we perform the migration manually in our approach, we required the system we used to be of manageable size to complete the study within a reasonable time frame.

The SCARF web application enables research conference organizers to create a discussion forum for papers submitted at a conference. It was originally developed at Stanford University for the SIGCOMM conference. SCARF is implemented in PHP and has a three-tier architecture where the application runs on a server and interacts with a database, while users access the application through a web browser. SCARF has been implemented to work with the MySQL database which is a relational SQL-based database. In this case study, we migrate it to use a document-oriented NoSQL database instead. We chose MongoDB for use as the document-oriented NoSQL database in this case study because it is one of the most popular NoSQL databases in use today [2].

We carried out the migration both in the original implementation as it exists as well as using our approach. The migration process consists of three steps: 1) schema migration, 2) data migration, and 3) code migration.

Schema migration: First, the schema that will be used in the NoSQL database has to be determined. Although NoSQL databases do not require a schema to be explicitly defined, application developers still need to decide how the data in the NoSQL database will be structured because it has an impact on query performance. So, to migrate the SCARF application to MongoDB, we have to migrate the schema used in the MySQL database.

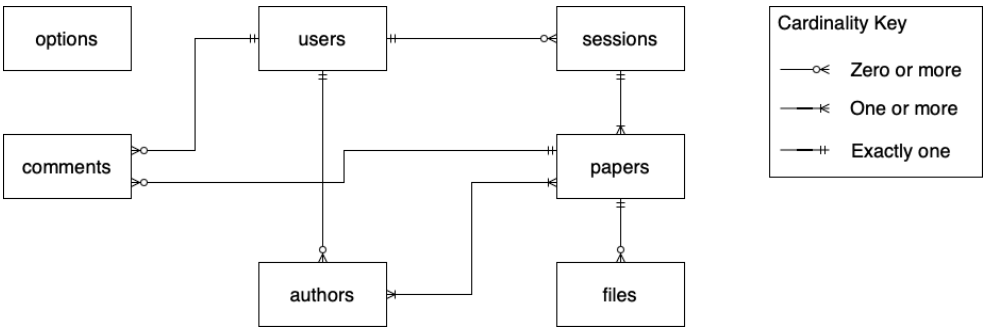


Figure 5.2: Data model showing the cardinality relationships among the tables in the database used in the SCARF application

Figure 5.2 shows the data model used in SCARF depicting the tables in the database and the cardinality relationships among them. As can be seen in the figure, there are seven tables in the

MySQL database used by SCARF. We need to determine how the data is these seven tables are going to be structured in the MongoDB database.

We have two options for the schema to be used when migrating an application from a relational database to a NoSQL database:

1. **Migration with no schema changes:** In this case, the structure of the data in the relational database is preserved when it is moved into the NoSQL database. This might be done when, for example, the migration of an application to a NoSQL database is carried out incrementally [61].
2. **Migration with schema changes:** In this case, the way the data is structured in the NoSQL database is different from the way it was originally structured in the relational database. This is usually done to improve query performance.

In this case study, we carried out the migration of SCARF from MySQL to MongoDB using both of the schema migration alternatives that we listed above. We created two MongoDB databases. In one database, the schema is the same as in the MySQL database and in the other, it's different.

To migrate SCARF to MongoDB without changing the schema used in its MySQL database, we have to specify how the tables, rows and columns in the MySQL database will be mapped to structures supported by MongoDB. In MongoDB, data is stored as *documents*. A document can be thought of as a hash table or a dictionary with an ordered set of keys, each of which has an associated value. Documents are stored in groups called *collections* [33]. To migrate SCARF to MongoDB without any changes to the schema used in the MySQL database, we created a collection corresponding to each table. The rows in a table were then stored as documents in the corresponding collection and the columns were mapped to keys in the documents.

When the database schema is changed during the migration of an application to a NoSQL database, one or more tables are *denormalized* [33]. In a relational database, tables are normalized to avoid redundancy in storing data. This is done by means of references. When a piece of information (such as the name of the reporting manager of the employees in an organization) may be repeated in multiple rows of a table, a new table is created to hold that information and references to rows in the new table are stored in the rows of the first table. When information from both tables needs to be combined in the results of a query, a *join* needs to be performed. Joins can slow down the execution of a query. So, tables are denormalized when they are migrated to a NoSQL database so that join operations can be avoided and query performance is improved. In MongoDB, denormalization is achieved by *embedding* a document in another document, whereby the document that's embedded is stored as the value of a key in the document in which it's being embedded. This is similar to repeating the same information in multiple rows of a table in a relational database.

Additionally, tables used for managing user permissions may not be migrated to the NoSQL database at all. This would also result in a different schema in the NoSQL database compared to the one used in the relational database.

In the case where we migrated SCARF to MongoDB and changed the schema, we denormalized the `papers` and `files` tables (see figure 5.2) because certain queries in the application involve joins between those two tables. We did this by embedding documents corresponding to the rows in the `files` table in documents corresponding to the rows in the `papers` table. Moreover,

the `users` table is used to manage user permissions. So, we retained it in the MySQL database and didn't move it to the MongoDB database.

Data migration: Once the schema to be used in the NoSQL database has been decided upon, the data in the relational database has to copied into the NoSQL database. To do this, schema changes, if any, have to be taken into consideration. In other words, the data should be copied into the NoSQL database in such a way that it conforms to the schema that's been decided to be used in the NoSQL database.

For this case study, we created a database in MySQL and populated it with synthetic data. We inserted data about 2085 users, 217 papers and 53 sessions into the database. The number of users, papers and sessions we selected is reflective of a typical research conference. We also inserted 2315 comments about the papers into the database.

We migrated the data in the MySQL database to both the MongoDB databases we created, the one in which we used the same schema as in the MySQL database and the one in which we used a changed schema. We wrote scripts to automate this process.

Code migration: The final step in the process of migrating an application from a relational database to a NoSQL database is changing the code so queries are executed on the NoSQL database instead of the relational database.

We changed the original implementation of the SCARF application and created two versions of the application. In one version, we changed the code to execute queries on the MongoDB database with the same schema as the MySQL database and in the other version, we changed it to execute queries on the database with the changed schema.

To evaluate the effectiveness of our approach in making the task of connector evolution easier when migrating an application from a relational database to a NoSQL database, we reimplemented SCARF using our architecture-centric approach[3]. For pragmatic reasons, we left the non-database related PHP code in each component unchanged, but wrapped the component in a WYVERN module using the foreign function interface of the language. Code in the components pertaining to executing queries and retrieving the results was replaced with calls to the database made through port interfaces in WYVERN. We had two options for the definition of the port interfaces:

1. **Abstract port interfaces:** In this case, the port interfaces are defined in an abstract, database-agnostic way. This enables the same interface to be used for executing queries on both the relational database as well as the NoSQL database. So, these port interfaces don't have to be changed when migrating the application to use a NoSQL database.
2. **SQL-specific port interfaces:** In this case, the port interfaces are designed to allow the execution of SQL queries only. The SQL queries to be executed are provided as arguments to the methods in the port interfaces. So, these port interfaces allow the components to interact only with relational SQL-based databases. As a result, these port interfaces have to be changed when migrating the application to use a NoSQL database.

[3]We do not consider this reimplementation as part of the cost of our approach, since if our approach is adopted, systems will be implemented this way from the beginning.

We reimplemented SCARF in WYVERN using both the alternatives for the definition of port interfaces. We created two versions of SCARF in WYVERN. In one version, we implemented the components to use the abstract port interfaces and in the other, we implemented them to use the SQL-specific port interfaces.

To finish the implementations in WYVERN, we also wrote an architecture specification in our ADL for each of the two versions. Additionally, we implemented connector abstractions that modularize the connectors that enable interaction with a relational database. We have two different implementations of the connector abstraction in the two versions. The two versions of the connector abstraction differ in the interfaces they require ports to have if the ports are to be attached to an instance of the modularized connector. In one version, the connector abstraction expects the abstract port interface and in the other, it expects the SQL-specific port interface. Note that implementation of the connector abstractions is a one-time effort similar to the implementation of a database library and can be reused for subsequent implementations of other systems.

We then migrated the two WYVERN implementations of SCARF to MongoDB. We carried out the migration of each implementation in two ways. We first migrated each of the two implementations to use the MongoDB database with the same schema as the MySQL database used in the original implementation of SCARF. Next, we migrated them to use the database with the changed schema. We thus have two migrated instances of the implementation with the abstract port interfaces and two of the implementation with the SQL-specific port instances. So, in all, we have four cases where we carried out the migration using our approach:

1. Migration of the implementation with the abstract port interfaces to the database with an unchanged schema
2. Migration of the implementation with the abstract port interfaces to the database with a changed schema
3. Migration of the implementation with the SQL-specific port interfaces to the database with an unchanged schema
4. Migration of the implementation with the SQL-specific port interfaces to the database with a changed schema

To carry out the migration, we implemented the connector abstraction that modularizes the connector we'd be using to interact with the MongoDB database. Again, this is a one-time effort analogous to a reusable implementation of a MongoDB library. We implemented the abstraction to work with the abstract port interfaces that we had designed when implementing the SCARF application in WYVERN. Since in the migration cases 1 and 2 mentioned above, the WYVERN implementation prior to the migration already uses the abstract interfaces, we continued to use them in the migrated versions too because we wanted to perform the migration with the least effort necessary. We decided to use these port interfaces in the migrated versions of the WYVERN implementation even in cases 3 and 4. This is because in these cases, it only matters that the port interfaces have to be changed for using the MongoDB database. The particular port interface that's used with MongoDB would not affect the results. As a result, we have used the same connector abstraction in the migrated versions in each of the four migration cases we listed above.

To finish the migration, we changed the architecture specification in each case to use the connector abstraction for the connector to be used with MongoDB. Furthermore, in each case,

we modified the architecture specification to include the MongoDB database component and connect the non-database components to it. Additionally, in cases 2, 3 and 4, we had to change the component code as well.

Results

We will now analyze the effectiveness of our approach in making the task of connector evolution easier when migrating an application from a relational database to a NoSQL database. We will do this by comparing the extent of the changes we made in each migration case of the WYVERN implementation of SCARF with the extent of the changes that we made in the corresponding case of the original PHP implementation.

The original implementation of the SCARF application consists of 13 components apart from the MySQL database that the non-database components interact with. These 13 non-database components are implemented in 19 PHP files which contain a total of 1,686 lines of code (including comments). There are a total of 88 SQL queries that are executed on the MySQL database.

Recall that we migrated the original implementation of SCARF to MongoDB in two ways. We first migrated it to use a MongoDB database which had the same schema as the MySQL database that was originally used. Then we migrated it to use a MongoDB database with a changed schema.

In the migration to the database without any schema changes, we had to change all 13 components. Out of the 19 PHP files, we had to change 17 files as they contain SQL queries. All of the 88 SQL queries had to be changed to equivalent MongoDB queries. In all, we had to change 221 lines in this case which constitute 13.1 percent of the total lines of code.

To migrate the original implementation of the SCARF application to the MongoDB database with the changed schema, we again changed all 13 components as well as 17 of the 19 PHP files. In this case, we retained the `users` table, which is used for managing user permissions, in the MySQL database. So, only 66 of the 88 SQL queries had to be changed as only those 66 queries involved tables that had been moved to the MongoDB database. We changed a total of 183 lines in this case which make up 7.4 percent of the total lines of code.

Recall also that we created two reimplemented versions of SCARF using our approach in WYVERN. In one version, we used abstract port interfaces in the components for making calls to the database. In the other version, we used SQL-specific port interfaces.

Like the original implementation, both reimplemented versions also consist of 13 components. The 13 components in each version are implemented using 35 files. This is roughly double the number of files in the original implementation, which has 19 files. This is due to the fact that we didn't implement the components natively in WYVERN. Instead, we kept the non-database related PHP code in the original implementation of each component unchanged and wrapped it in a WYVERN module. So, we have an extra file for each component in which we implement the wrapper WYVERN module. Additionally, we have an architecture specification file and a couple of files in which we implement auxiliary functions. As a result, the number of files in the WYVERN versions of SCARF is nearly double the number of files in the original implementation.

As in the original implementation, there are 88 locations in each reimplemented version where queries are executed on the database. These queries are executed through component ports rather than by direct invocation of database library functions.

The WYVERN version in which we have abstract port interfaces is implemented using a total

Table 5.7: Comparison between the changes made to the WYVERN implementation of SCARF with the abstract port interfaces and the changes made to the original implementation to migrate them to the MongoDB database with an unchanged schema

	Original implementation	**WYVERN**
No. of non-database components	13	13
No. of files	19	35
No. of lines	1686	2684
No. of queries	88	88
No. of non-database components changed	13	0
No. of files changed	17	1
No. of queries changed	88	0
No. of lines changed	221	16
Percentage of lines changed	13.1%	0.5%

2,684 lines of code, whereas the version with the SQL-specific port interfaces has 2,831 lines of code in total. Again, this increase in the number of lines of code in the WYVERN versions is explained by the fact that we have not implemented the components natively in WYVERN as well as the fact that we have an architecture specification and files for auxiliary functions.

As discussed earlier, we migrated each reimplementation of SCARF in WYVERN to MongoDB in two ways. We migrated both versions to use a MongoDB database in which the schema was the same as in the original MySQL database. We also migrated them to a MongoDB database with a changed schema. This gives us four migration cases in WYVERN. Below, we will compare the extent of the changes made in each of the four migration cases of the WYVERN implementation of SCARF with those made in the corresponding case of migrating the original implementation.

Case 1: Migration of the implementation with the abstract port interfaces to the database with an unchanged schema

As can be seen in table 5.7, none of the non-database components had to be changed when migrating the WYVERN implementation of SCARF with the abstract port interfaces to the MongoDB database with the same schema as the original MySQL database. We only had to change the architecture specification file. We had to change the connector type used from the one for MySQL databases to the one for MongoDB databases. Additionally, we had to change the database component type from the MySQL one to the MongoDB one. All this involved modifying only 16 lines in the architecture specification. None of the query-related code had to be changed as queries are executed through component ports and in this case, ports have a database-agnostic interface which can be used to execute queries on both MySQL and MongoDB databases. Therefore, we had to change just 0.5 percent of the lines and all these lines were in the architecture specification file.

In contrast, in the original implementation, we had to change 221 lines (13.1 percent of total lines) to migrate it to the MongoDB database with the same schema as the original MySQL database. This is because all of the 88 SQL queries in the original implementation had to be modified to equivalent MongoDB queries. And these modifications had to be made in 17 of the

Table 5.8: Comparison between the changes made to the WYVERN implementation of SCARF with the abstract port interfaces and the changes made to the original implementation to migrate them to the MongoDB database with a changed schema

	Original implementation	**WYVERN**
No. of non-database components	13	13
No. of files	19	35
No. of lines	1686	2684
No. of queries	88	88
No. of non-database components changed	13	13
No. of files changed	17	31
No. of queries changed	66	11
No. of lines changed	183	104
Percentage of lines changed	7.4%	3.9%

19 files, resulting in changes to all 13 components.

So, in this case, carrying out the migration in the WYVERN implementation is much easier than performing the migration in the original implementation.

Case 2: Migration of the implementation with the abstract port interfaces to the database with a changed schema

Table 5.8 shows the changes we made to the WYVERN implementation of SCARF with the abstract port interfaces to migrate it to the MongoDB database in which the schema had been changed from the one used in the original MySQL database. In this case, we had to change all 13 non-database components. Since the `users` table is retained in the MySQL database, the migrated application interacts with both MySQL and MongoDB databases. In fact, each non-database component interacts with both databases after the migration. So, we added an additional requires port to each of those components to enable them to execute queries on the MongoDB database. Recall from section 3.3.2 that a requires port is reified via an argument in the definition of the WYVERN module corresponding to the component to which the port belongs. So, we changed the module definitions corresponding to the non-database components in the SCARF application and added a requires port argument that would be used by the components to make calls to the MongoDB database.

Of the 88 queries present in the application, 22 involve only the `users` table. Since the `users` table is retained in the MySQL database and the port used for interacting with the MySQL database in each component is left unchanged, we didn't have to change the code for executing these 22 queries at all. Of the remaining 66 queries, we had to change the queries themselves in the case of 11 queries. In the original implementation, these 11 queries each involve a join between the `users` table and a (varying) table that's moved to the MongoDB database during the migration. So, in the migrated version, the queries are changed to perform a join between a MySQL table and a MongoDB collection. In the remaining 55 queries, only tables that are moved to the MongoDB database are involved. Since, in the migrated code, we use the same abstract interface for executing queries on the MongoDB database as was used in the code before

the migration for interacting with the MySQL database, the queries themselves don't have to be changed. However, the ports used for executing those queries have to be changed because the queries have to be executed on the MongoDB database after the migration. So, we changed the code for executing each of these 55 queries to use the requires port argument of the module in which the query appears that's for the MongoDB database instead of the one for the MySQL database. These modifications constitute a simple variable change. To sum up, in the WYVERN implementation in this case, the code for executing 22 queries wasn't changed at all, modifying the code for executing 55 queries involved making simple variable changes and modifying the code for executing the remaining 11 queries involved making more complex changes because the queries themselves had to be changed.

In the original implementation too, the code for executing the 22 queries involving only the `users` table didn't have to be changed at all when migrating to the MongoDB database with the changed schema. This is because, as mentioned above, the `users` table is retained in the MySQL database due to which the code for executing these queries on the MySQL database can be continued to be used even after the migration. In the case of the remaining 66 queries, the queries themselves had to be modified. These queries involve tables that are moved to the MongoDB database. So they need to be run on the MongoDB database after the migration instead of on the MySQL database. Since they are specified using SQL in the original implementation, they have to be transformed into equivalent MongoDB queries. Therefore, modifying the code for these 66 queries involved making complex changes as the queries themselves were changed. To make these modifications, we had to change the implementation of all 13 non-database components in the original implementation of the SCARF application.

In WYVERN, we also changed the architecture specification and added a component type declaration and a component instance declaration for the MongoDB database. We also modified the component type declaration for each non-database component and included an additional requires port to be used for interaction with the MongoDB database. Additionally, we added a connector type declaration and connector instance declarations for the connectors to be used for executing queries on the MongoDB database. Furthermore, we added attachments to connect the non-database component instances to the instance of the MongoDB database component. To make the above changes, we had to add 47 new lines to the architecture specification.

In all, we had to change 104 existing lines (3.9 percent of all lines) in the WYVERN implementation in this case. As mentioned above, we also added 47 new lines to the architecture specification. All these changes were spread across 31 files. By contrast, in the original implementation, we changed 183 existing lines (7.4 percent of all lines) which were spread across 17 files. Note that the number of files modified in the WYVERN implementation is roughly double the number of files modified in the original implementation due to the fact that we have not implemented the components natively in WYVERN and have an extra file for each component in which we implemented the wrapper module for the component. We modified both the files containing the wrapper modules as well as the files containing the adapted PHP code for the components. Additionally, as noted above, we changed the architecture specification file as well. Because of all this, the number of files modified in the WYVERN implementation is nearly double the number of files modified in the original implementation.

Unlike case 1, the changes made in the WYVERN implementation in this case are not completely localized to the architecture specification. There are two reasons for this. First, the ports

in the components were changed. In particular, a new port for interacting with the MongoDB database was added to each non-database component. Second, the operations performed through individual ports were changed as well. Specifically, in any component, some or all of the operations in the execution of a query involving a table moved to the MongoDB database are performed using the newly added port for the MongoDB database after the migration. Before the migration, these operations were being performed using the port for the MySQL database. So, some of the operations performed through the ports for the MySQL database before the migration are changed to be performed using the ports for the MongoDB database after the migration. Due to the two reasons mentioned above, we had to change the code for the components as well in this case. So, the changes weren't localized just to the architecture specification.

Even though the changes we made in the WYVERN implementation in this case are not wholly localized to the architecture specification, they are simpler and less extensive than those we made in the original implementation. This is due to the fact that we didn't change the interface for the database ports in the non-database components. As a result, we didn't have to change the code for the execution of most queries.

We had to carry out complex changes in the WYVERN implementation only for the 11 queries (out of 88) containing a join involving the `users` table. This is because the way these queries are executed has to be changed for migrating to the MongoDB database. Before the migration, since both tables in the join are stored in the MySQL database, the join can be performed in the MySQL database server; the tables to be used for performing the join and the join condition have to be sent to the MySQL database server when issuing the query. During the migration, the `users` table is retained in the MySQL database, while the other tables involved in the joins are moved to the MongoDB database. So, to execute a query containing a join involving the `users` table after the migration, the data in the `users` table and that in the MongoDB collection to which the other table in the join has been moved have to be retrieved separately, and the join has to be implemented in the code for the SCARF application. Modifying the code in this way is a complex change. In the original implementation too, modifying the code for executing these 11 queries is a complex change because we had to make similar changes.

We didn't have to change the code for the 22 queries involving only the `users` table in the WYVERN implementation. Since the `users` table was retained in the MySQL database, we could continue to use the ports for the MySQL database for executing these queries even after the migration. Moreover, we didn't change the interface for these ports during the migration. So, the way the execution of these queries was implemented using the port interface methods didn't have to be changed either. In the original implementation too, we didn't have to change the code for executing these 22 queries. This is because it's implemented using a MySQL database library which can be continued to be used even after the migration as the `users` table is retained in the MySQL database.

For the remaining 55 queries, which involve only tables that are moved to the MongoDB database, we only had to make simple variable changes in the code for executing these queries in the WYVERN implementation. Before the migration, the queries are executed using the ports for the MySQL database. So, they have to be changed to use the ports for the MongoDB database, which is a simple variable change. Since we have used the same interface for the ports for the MySQL database as well the ports for the MongoDB database, no other change needs to be made to the code for executing these queries. In the original implementation, however, SQL is used to

Table 5.9: Comparison between the changes made to the WYVERN implementation of SCARF with the SQL-specific port interfaces and the changes made to the original implementation to migrate them to the MongoDB database with an unchanged schema

	Original implementation	**WYVERN**
No. of non-database components	13	13
No. of files	19	35
No. of lines	1686	2831
No. of queries	88	88
No. of non-database components changed	13	13
No. of files changed	17	31
No. of queries changed	88	88
No. of lines changed	221	276
Percentage of lines changed	13.1%	9.7%

specify these queries. So, they have to be transformed into equivalent MongoDB queries, which is a complex change. So, all in all, the changes made in the WYVERN implementation are simpler and less extensive than those made in the original implementation due to the fact that we used the same interface for the database ports before and after the migration.

An interesting thing to note here is that unlike case 1, where connector instances of one type are replaced with instances of another connector type, in this case, new connector instances of a different type are added and existing connector instances are left unchanged.

Case 3: Migration of the implementation with the SQL-specific port interfaces to the database with an unchanged schema

Table 5.9 shows the changes we made to the WYVERN implementation of SCARF in which we used SQL-specific port interfaces to migrate it to the MongoDB database in which we used the same schema as the original MySQL database.

In this case, we made changes to each of the 13 non-database components in the WYVERN implementation. This involved changing the interfaces for the ports used for making calls to the database as well as changing the code for executing queries.

In each component, we changed the interface for the port used by the component to interact with the database. We changed the interface from the SQL-specific interface to the abstract, database-agnostic interface so the port could be used to make calls to the MongoDB database after the migration. This involved changing both the code for the non-database components as well as the architecture specification. In the architecture specification, we modified the component type declarations for the non-database components. In particular, in each component type for a non-database component, we changed the interface for the port meant for interacting with the database to the abstract, database-agnostic interface. To change the interfaces of the ports in code, we had to change the definition of the WYVERN module corresponding to each non-database component. Specifically, in each module definition, we changed the type of the argument that corresponds to the port used for interacting with the database from the SQL-specific interface's type to the abstract interface's type.

In this case, the code for executing each of the 88 queries had to be changed in the WYVERN implementation. This is because, as mentioned above, we changed the port interfaces during the migration. As a result, the code for issuing the queries using the ports with the SQL-specific interfaces had to be transformed into equivalent code that issued them using the ports with the abstract interfaces. This was an extensive and complex change.

Finally, in the architecture description, we changed the connector type used from the one for MySQL databases to the one for MongoDB databases. We also changed the database component type from the one for the MySQL database to the one for the MongoDB database.

Thus, in the WYVERN implementation in this case, we changed a total of 276 lines which were spread across 31 files. That is 9.7 percent of the total lines of code.

The changes made in this case in the WYVERN implementation are similar to those made in the original implementation. In the original implementation too, we changed each of the 13 components. Also, each of the 88 SQL queries had to be transformed into equivalent MongoDB queries. Thus, we had to change the code for executing all 88 queries. To carry out these complex and extensive changes in the original implementation, we had to change 221 lines which were spread across 17 files. So, in the original implementation, we changed 13.1 percent of total lines.

Note that, as in case 2, the number of files modified in the WYVERN implementation in this case too is roughly double the number of files modified in the original implementation. Again, this is due to the fact that we haven't implemented the components natively in WYVERN.

In this case, the changes in the WYVERN implementation are as widespread as the changes in the original implementation. This is because the port interfaces in the WYVERN implementation are changed. As a result, every location in the implementation where the old interface is used has to be changed to use the new interface. Moreover, since the two interfaces have different methods, the functionality implemented using the methods in the old interface has to be transformed and implemented using the methods in the new interface. This is similar to changing a library which is what is done in the original implementation where query execution implemented using the MySQL library is changed and implemented using the MongoDB library. As a result, the extent of the changes made in the WYVERN implementation in this case is the same as the extent of the changes in the original implementation.

Case 4: Migration of the implementation with the SQL-specific port interfaces to the database with a changed schema

Table 5.10 shows the changes we made to the WYVERN implementation of SCARF with the SQL-specific port interfaces to migrate it to the MongoDB database with a different schema than the one used in the original MySQL database.

In this case too, we changed all 13 non-database components in the WYVERN implementation. As in case 2, since the `users` table is retained in the MySQL database, each non-database component interacts with both the MySQL database as well as the MongoDB database after the migration. So, we added an additional requires port to each non-database component to enable it to make calls to the MongoDB database. To do this, we changed the component type declaration for each non-database component in the architecture specification and added a new requires port to it that could be used for interacting with the MongoDB database. We declared the new ports to use the abstract, database-agnostic interface that we've designed. We also changed the definition

Table 5.10: Comparison between the changes made to the WYVERN implementation of SCARF with the SQL-specific port interfaces and the changes made to the original implementation to migrate them to the MongoDB database with a changed schema

	Original implementation	**WYVERN**
No. of non-database components	13	13
No. of files	19	35
No. of lines	1686	2831
No. of queries	88	88
No. of non-database components changed	13	13
No. of files changed	17	31
No. of queries changed	66	66
No. of lines changed	183	207
Percentage of lines changed	7.4%	7.3%

of the WYVERN module for each non-database component and added an additional argument in each definition that reifies the requires port for the MongoDB database added in the component type corresponding to the non-database component. This resulted in a change to the code of all 13 non-database components.

Furthermore, we had to change the code for the execution of 66 of the 88 queries. Of the 66 queries, 11 queries involve a join between the `users` table, which is retained in the MySQL database, and a (varying) table that's moved to the MongoDB database. So, as in case 2, we changed the code for running these queries to execute them by making calls to both the MySQL database and the MongoDB database. The remaining 55 queries do not involve the `users` table at all; all the tables involved in these queries are moved to the MongoDB database. We made two changes to modify the code for executing these 55 queries. First, we changed the port used to execute them. In the initial WYVERN implementation in this case, the queries are executed using the ports for the MySQL database. We changed the code so that the ports for the MongoDB database will instead be used for executing these queries. Second, we changed the way the queries are issued as well. In this case, we use a different interface for the ports for the MongoDB database than the interface we use for the MySQL database. In particular, we use the SQL-specific interface for the ports for the MySQL database, whereas we use the abstract, database-agnostic interface for the ports for the MongoDB database. So, we changed the code for the execution of the 55 queries to make calls to the MongoDB database using the methods in the abstract interface, where calls were being made to the MySQL database using the methods in the SQL-specific interface prior to the migration. All these modifications were complex changes.

We didn't have to change the code for the execution of the remaining 22 out of the 88 queries at all. These queries involve only the `users` table. Since the `users` table is retained in the MySQL database, the code for executing these 22 queries didn't have to be changed at all.

In the original implementation too, we didn't have to change the code for the execution of the 22 queries involving `users` table. We had to change the code for the execution of the remaining 66 queries to make calls to the MongoDB database. All these changes resulted in modifications to all 13 non-database components.

As in case 2, we changed the architecture specification in WYVERN in this case too to add a component type declaration and a component instance declaration for the MongoDB database. We also added a connector type declaration and connector instance declarations for the connectors to be used for making calls to the MongoDB database. Lastly, we added attachments to connect the non-database component instances to the instance of the MongoDB database component. As in case 2, to make the changes to the architecture description, including the changes to add a new port to each component type corresponding to each non-database component, we added 47 new lines to the architecture specification.

In total, we changed 207 existing lines in the WYVERN implementation. These changes were spread across 31 files. Similarly, in the original implementation, we changed a total of 183 lines which were spread across 17 files. Again, the number of files modified in the WYVERN implementation is roughly double the number of files changed in the original implementation because we didn't implement the components natively in WYVERN. In the original implementation, we changed 7.4 percent of the total lines of code, whereas in the WYVERN implementation, we changed 7.3 percent of the total lines of code.

As in case 3, the changes made in the WYVERN implementation in this case too are similar to those made in the original implementation. Again, as explained in the discussion of case 3, this is due to the fact that we use a different interface for the ports for the MongoDB database than the one used for ports for the MySQL database.

As in case 2, here too, existing connector instances are left unchanged and new connector instances of a different type are added.

Limitations

In this case study, we focus solely on evaluating the ease with which the SCARF application can be migrated to use a MongoDB database with our approach. We aren't concerned about query performance at all in this study. In particular, we haven't analyzed whether the different schema alternatives we selected for the MongoDB database provide improved query performance after the migration. We simply selected these alternatives based on the suggestions commonly made in the research literature as well as in NoSQL migration guides.

Threats to Validity

With respect to external validity, we note that we have evaluated our approach on an application of fairly small size. Specifically, the application we have used in this study, SCARF, consists of 1,686 lines of code. However, we believe our results generalize to larger systems too. This is because database calls are implemented the same way in larger systems as they are in SCARF. For example, the largest system we found in the literature on which a SQL to NoSQL migration study has been performed is Wordpress[4], which is a content management system that can be used to build websites. Wordpress is implemented using about 550,000 lines of code, which are split into 1,158 files. As in SCARF, the database interactions in Wordpress too are implemented as SQL queries which are executed using the PHP MySQL library. There are about 500 queries in Wordpress which are spread across 78 files. This is similar to SCARF where the queries are

[4]https://wordpress.org

spread across multiple files as well. As a result, we believe that our results would generalize to larger systems as well.

With respect to construct validity, we note that we have used number of lines of code as a measure of the difficulty of performing the task of connector evolution. However, there might be other factors that have an influence on the difficulty of the connector evolution task. It might be possible to uncover those factors only in a real development setting. In other words, our measure of effort might only be an approximation and a true estimate would require observing expert developers using our approach in their projects.

5.1.3 Discussion

Taken together, the ROS 1 to ROS 2 migration case study as well as the SQL to NoSQL migration case study show that our approach, which involves the integration of the architecture description of a system with the code, makes the task of connector evolution much easier in cases where the port interfaces need not be changed. In these cases, the changes to be made are completely localized to the architecture description. This can be seen in the entire ROS 1 to ROS 2 migration case study. In the SQL to NoSQL migration case study, we see this in the case where we migrate the implementation of the SCARF application based on our approach in which we use the abstract, database-agnostic port interfaces to the MongoDB database in which we use the same schema that was used in the MySQL database (case 1).

Even in cases where new ports have to be added to one or more components to perform connector evolution, our approach can make the task easier if the resulting modifications in our approach only involve offloading some of the communication operations performed using the existing ports to the new ports. In this case, our approach makes the task easier if the following two conditions hold for each offloaded operation: 1) the old port using which an operation is performed and the new port to be used for it have the same interface, and 2) the points in the execution of a component where the operation is performed aren't changed. In this case, the changes aren't completely localized to the architecture description. The code for one or more components would have to be changed as well. However, those modifications involve just changing the ports to be used for each offloaded operation from the old one that was used for it to the new one. These modifications constitute simple variable changes. This can be seen in the case in the SQL to NoSQL migration case study where we migrate the implementation of SCARF in WYVERN that uses the abstract, database-agnostic port interfaces to the MongoDB database with a different schema than the original MySQL database (case 2).

When port interfaces are changed during connector evolution, our approach doesn't provide any benefit over current implementation approaches. In this case, complex changes would have to be made throughout the code as if the connectors had been implemented by directly using a communication library. This is seen in the SQL to NoSQL case study in the two cases where we migrated the implementation of SCARF in WYVERN that uses the SQL-specific port interfaces (cases 3 and 4).

Applicability of the Connector Evolution Approach Implemented in WYVERN

Our connector evolution approach aids only with the task of migrating connectors in a system. In many cases, modifying the architecture of a system only involves switching one or more connectors in the system with different ones. This occurs, for example, when changing from RabbitMQ to Apache Kafka, or from Netflix Hystrix to resilience4j. In such scenarios, using our approach will help restricting the changes to be made to the architecture specification if the port interfaces have been designed at an appropriate level of abstraction. In other situations, however, modifying the architecture might involve changing one or more components in addition to replacing one or more connectors. This may occur, for example, when changing from a SQL-based database to a NoSQL database. The tasks involved in changing components may vary depending on the type of the component. For example, to replace a database component, an appropriate schema has to be designed and the data in the original database has to be migrated to the new database. Our connector evolution approach doesn't help with such tasks. However, when a component is replaced, a new connector might have to be used to interact with the new component. This occurs, for example, in our SQL to NoSQL migration case study, where a new connector has to be used when the SQL-based database is replaced with the NoSQL database. In situations like this, our approach can help limit the changes to be made to the architecture description if the component interfaces aren't changed.

Costs Associated with the Connector Evolution Approach in WYVERN

In our two case studies, we have evaluated how our connector evolution approach aids application developers in carrying out the task of changing connectors. We have assumed that connector abstractions are available for use by application developers and that developers are comfortable using them. If these two assumptions are met, then our approach makes the task of connector evolution easy in cases where component interfaces aren't changed by restricting the modifications to be made to the architecture specification.

The task of designing the connector abstractions themselves, however, can be quite involved. If a connector abstraction is to be designed and implemented from scratch, then it can take a great deal of effort, similar to the implementation of a framework like ROS or Spring. In our two case studies, though, we implemented the connector abstractions using already existing frameworks and libraries (specifically, the ROS 1 and ROS 2 frameworks, and libraries for MySQL and MongoDB databases). Implementing the connector abstractions using existing frameworks and libraries is relatively straightforward. As explained in section 3.3.2, connector abstractions are implemented using WYVERN types. To define the WYVERN type associated with a connector, metadata needs to be added to the type. In particular, methods with predefined names and signatures need to be defined in the metadata to implement any custom typechecking for the ports attached to the connector as well as to generate the code that initializes the connector and implements the connector's functionality. To implement a connector abstraction, we first designed the port interfaces that it would support. This requires careful thought to determine the details that should be hidden. We then implemented the metadata methods for typechecking and code generation. This is straightforward if the port interfaces have already been designed and an existing framework or library is used for the implementation.

Application developers might have to pay a cost too in getting accustomed to the way a system is implemented in WYVERN. In particular, in WYVERN, components are implemented to interact via interfaces. Direct use of communication libraries to implement component interactions is prohibited. This might require cognitive effort on the part of application developers to get used to as they need to think about component interactions at an abstract level rather than about low-level details of the interaction.

Limitations of the Connector Evolution Approach in WYVERN

Our connector evolution approach only helps with isolating changes associated with the implementation of component interactions to the architecture description. In many systems, a framework, such as Spring Boot, is used to implement communication among components. Many frameworks provide other services in addition to communication services. For example, the Spring Boot framework provides security services (such as authentication and access control), application health monitoring services and logging services in addition to providing support for the implementation of various communication mechanisms. Use of such services provided by a framework is also usually widely scattered across the codebase. So, when a framework is to be changed, locations in the code where any of the services provided by the framework have been used need to be changed as well. Our connector evolution approach, however, only helps in making the task of changing code related to the implementation of connectors easy. It doesn't help with changing code related to the use of non-communication related services provided by the framework.

5.2 Claim 2: Generality of the Connector Abstraction Mechanism and the Connector Evolution Approach

We evaluated the expressiveness of the connector abstraction mechanism we developed in our connector evolution approach by implementing a wide range of connectors using our abstraction mechanism. We also evaluated the generality of our connector evolution approach by showing how it can be used to change connectors in a wide range of scenarios. We outline the individual connectors and connector evolution scenarios that we implemented in the subsections below.

5.2.1 Generality of the Connector Abstraction Mechanism

As explained in section 3.3.2, in our approach, we implement connector abstractions using WYVERN types. In particular, corresponding to a connector type that can be used in the architecture description, a WYVERN type needs to be defined to modularize the connector's implementation. Modularization of a connector, as explained previously, is achieved via metadata that's added to the WYVERN type for the connector. In particular, methods with predefined names and signatures need to be defined in the metadata of the WYVERN type for the connector to implement any custom typechecking for the ports attached to the connector as well as to generate the code that initializes the connector and implements the connector's functionality.

We evaluated the expressiveness of our connector abstraction mechanism by implementing a wide range of connectors using the abstraction mechanism. We used Mehta et al.'s taxonomy

of connectors [75] to guide our selection of connectors to implement. The taxonomy classifies connectors into one of eight different types: 1) procedure call, 2) event, 3) data access, 4) linkage, 5) stream, 6) arbitrator, 7) adaptor, and 8) distributor. Below, we describe which connectors from each category we implemented.

Procedure Call

Procedure call connectors are used to transfer control from one component to another through various invocation techniques. They also enable transfer of data between the components through parameters and return values.

To demonstrate that procedure call connectors can be implemented using our connector abstraction mechanism, we implemented a connector that enables remote procedure calls (RPCs) to be made using the gRPC framework. The gRPC framework supports the implementation of both synchronous and asynchronous RPCs. So, we have implemented synchronous and asynchronous versions of the gRPC connector. RPC methods implemented using the gRPC framework must have exactly one parameter and a non-void return type. We have implemented this custom typechecking rule for gRPC methods in both versions of the gRPC connector.

As mentioned in section 5.1.1, we have also implemented connector abstractions for services and actions in each of the two generations of the Robot Operating System (ROS) framework, ROS 1 and ROS 2. Specifically, we have implemented a connector abstraction for services in ROS 1 and another for services in ROS 2. Similarly, we have implemented a connector abstraction for actions in ROS 1 as well as one for actions in ROS 2. These are all examples of procedure call connectors too. As explained previously, services in both ROS 1 and ROS 2 are a synchronous remote procedure call mechanism and actions are an asynchronous call/return mechanism.

Event

Event connectors enable the transfer of data and control through a notification mechanism in which the producers and consumers of an event need not be aware of each other's identities.

Using our connector abstraction mechanism, we have implemented two event connectors in which events are dispatched using message queues. In one of the two connectors, the RabbitMQ library is used for implementing the message queue. In the other, we have used the Apache Kafka library for implementing the message queue. The two connectors differ in the way consumers are notified of an event. In the RabbitMQ-based event connector, consumers get notified via callback methods, whereas in the Apache Kafka-based event connector, consumers must repeatedly poll the message queue to be notified of events. However, developers using the connector abstractions need not be concerned with these differences as they only need to implement components to interact using ports whose interfaces are specified in the architecture description. As explained in section 3.3.2, the code for tying the components together is generated by the connector abstraction.

In the ROS 1 to ROS 2 case study (see section 5.1.1), we implemented connector abstractions for topics in both ROS 1 and ROS 2. Topics in ROS are also examples of event connectors.

Data Access

Data access connectors enable components to access data kept in a data storage component, such as a relational database, a NoSQL database or a distributed file system. As explained in section 5.1.2, with our connector abstraction mechanism, we have implemented a data access connector that enables interaction with a MySQL database. We have also implemented a connector that enables interaction with a MongoDB database.

Linkage

Linkage connectors bind a name used in one module to the implementation provided by another module. Examples of linkage connectors include the C export mechanism and the Java dynamic class loader. The connector abstraction mechanism in our approach is designed to enable components to interact at run time, not resolve dependencies between compilation units. As a result, our connector abstraction mechanism doesn't support the implementation of linkage connectors.

Stream

Stream connectors are used to transfer a sequence of data between loosely coupled components. The pipe connector in a pipe-and-filter architectural style is an example of a stream connector. In practice, pipe connectors are often implemented using message queues [1]. So, we have implemented two pipe connectors using message queues with our connector abstraction mechanism. In one of the two connectors, we use the RabbitMQ library for implementing the message queue and in the other, we use the Apache Kafka library.

Arbitrator

Arbitrator connectors provide various mediation services, such as fault handling, load balancing and scheduling, for coordinating interaction among various components.

Using our connector abstraction mechanism, we have implemented two connectors that each handle failures in synchronous gRPC calls using the circuit breaker pattern. The circuit breaker pattern is used to interrupt calls to a failing component, thus reducing the load on the component and giving it time to recover [81]. A circuit breaker is implemented as a state machine (see figure 5.3). The state machine has three states: closed, open and half open. The circuit breaker starts in the closed state. In the closed state, the circuit breaker allows calls to go through. However, calls are monitored to determine whether they succeeded or failed. Once the failure rate crosses a configured threshold, the circuit breaker transitions to the open state. When the circuit breaker is in the open state, calls fail immediately, without being directed to the component that would handle them. After a certain (configurable) time period elapses, the circuit breaker transitions to the half-open state. In the half-open state, a certain (configurable) number of calls are allowed to be routed to the component that would handle them. Calls beyond the allowed number fail immediately. The calls that are allowed to go through in the half open state are monitored to determine whether they succeeded or failed. If the failure rate is greater than or equal to a (configurable) threshold (which may be different than the threshold configured for the

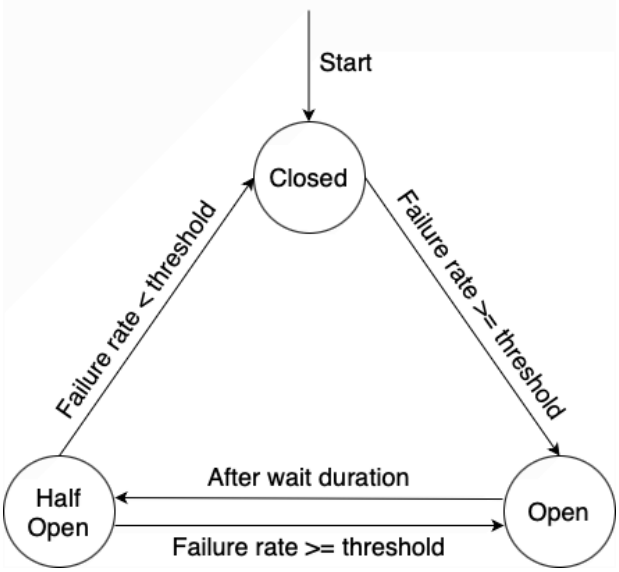


Figure 5.3: State machine for the operation of a circuit breaker

closed state), the circuit breaker goes to the open state. If the failure rate is below the threshold, it transitions to the closed state.

We have implemented an arbitrator connector that implements the circuit breaker pattern for synchronous gRPC calls using the Netflix Hystrix library. Netflix stopped active development of Hystrix in 2018. So, the resilience4j library has been developed as a replacement for Hystrix. We have also implemented an arbitrator connector that implements a circuit breaker for synchronous gRPC calls using the resilience4j library with our connector abstraction mechanism.

Adaptor

Adaptor connectors enable interaction between components that have not been designed to interoperate. These connectors may provide interoperability between components that use different interaction protocols, support interconversion between different data formats used by the interacting components or provide adaptation between components with different interfaces.

With our connector abstraction mechanism, we have implemented an adaptor connector that enables web service clients that use the Simple Object Access Protocol (SOAP) to invoke services that are exposed using a REpresentational State Transfer (REST) API. Traditionally, web applications have been implemented as a monolith. A monolithic web application might expose web services that may be invoked from other web applications. Many legacy web services use SOAP as the messaging protocol. When monolithic web applications are migrated to a microservice-based architecture, the web services they expose might be reimplemented as services that are exposed using a REST API. If there are clients that haven't been migrated, then an adaptor is needed to enable them to invoke the REST-based services even though they continue to use SOAP. We have used our connector abstraction mechanism to implement a SOAP to REST adaptor connector that enables legacy web service clients that use SOAP to invoke services that have been updated to provide a REST API.

Table 5.11: Connectors implemented using the connector abstraction mechanism in WYVERN

Type of connector	Connectors implemented
Procedure call	1. Synchronous gRPC connector 2. Asynchronous gRPC connector 3. ROS 1 service connector 4. ROS 2 service connector 5. ROS 1 action connector 6. ROS 2 action connector
Event	1. Event connector implemented using RabbitMQ 2. Event connector implemented using Apache Kafka 3. ROS 1 topic connector 4. ROS 2 topic connector
Data access	1. Connector for MySQL databases 2. Connector for MongoDB databases
Linkage	Not implemented because linkage connectors are not run-time entities
Stream	1. Pipe connector using RabbitMQ 2. Pipe connector using Apache Kafka
Arbitrator	1. gRPC connector with circuit breaker implemented using Netflix Hystrix 2. gRPC connector with circuit breaker implemented using resilience4j
Adaptor	1. SOAP to REST adaptor connector
Distributor	Not implemented because distributor connectors are low-level connectors

Distributor

Distributor connectors identify interaction paths between components and route communication along these paths. They enable components to be located based on symbolic names. Examples of distributor connectors include various network services, such as Domain Name Service (DNS), routing and switching. Since distributors are low-level connectors that only provide routing services to other connectors and are not first-class connectors themselves, we have not implemented any distributor connector using our connector abstraction mechanism.

Summary

We have demonstrated the generality of our connector abstraction mechanism by showing how it can be used to implement a wide range of connectors. Table 5.11 succinctly shows the connectors we have implemented from each category in Mehta et al.'s taxonomy of connectors using our connector abstraction mechanism.

5.2.2 Generality of the Connector Evolution Approach

We evaluated the generality of our connector evolution approach by showing how it can be used to change connectors in a number of scenarios drawn from the real world. As discussed in section 5.2.1, in WYVERN, we have implemented connectors from the different categories in

Mehta et al.'s taxonomy of connectors [75]. To show the generality of our connector evolution approach, we implemented connector evolution scenarios involving connectors in each category where we have implemented more than one connector. These scenarios are based on the types of connector evolution scenarios that developers are interested in today. Below, we describe the scenarios we implemented in each category.

Procedure Call

We implemented four scenarios in this category: 1) changing a ROS 1 service to a ROS 2 service, 2) changing a ROS 1 action to a ROS 2 action, 3) changing a ROS 1 service to a synchronous gRPC call, and 4) changing a ROS 1 action to an asynchronous gRPC call.

We implemented scenarios 1 and 2 in the ROS 1 to ROS 2 migration case study (see section 5.1.1). As mentioned previously, when changing a ROS 1 service to a ROS 2 service or a ROS 1 action to a ROS 2 action in a system implemented using our approach, the changes are localized to the architecture description. By contrast, when a ROS-based system implemented using existing programming languages is changed from ROS 1 to ROS 2, the changes are spread throughout the codebase.

Scenarios 3 and 4 are exemplars of situations where ROS software has to be migrated to a robotic platform, such as the Spot robot from Boston Dynamics, which does not support ROS. In software written for the Spot Robot, gRPC is used for interaction among the components. So, a ROS-based system needs to be changed to use gRPC if it is to be used on the Spot Robot.

For scenario 3, we used the system implemented in the tutorial on ROS services in the ROS wiki page [5]. We implemented the system in Python as well as in WYVERN. To change the Python implementation to use synchronous gRPC calls, we had to make changes throughout the code. In the WYVERN implementation, on the other hand, we made changes only to the architecture description.

Similarly, for scenario 4, we used the system implemented in the tutorial on ROS actions in the ROS wiki page [3, 4]. Again, we implemented the system in Python as well as in WYVERN. We changed the system to use asynchronous gRPC calls. In the Python implementation, again, we had to make changes throughout the code. In the WYVERN implementation, we had to change only the architecture description.

Event

We implemented two scenarios in this category: 1) changing a ROS 1 topic to a ROS 2 topic, and 2) changing a RabbitMQ-based event connector to an Apache-Kafka based event connector.

We implemented scenario 1 in the ROS 1 to ROS 2 migration case study (see section 5.1.1). As mentioned previously, when ROS 1 topics in a ROS-based system implemented in WYVERN are changed to ROS 2 topics, the changes need to be made only in the architecture description. In contrast, when ROS 1 topics in a system implemented using existing programming languages are to be changed to ROS 2 topics, widespread changes need to be made to the codebase.

For scenario 2, we implemented a producer/consumer system that has one producer and two consumers. The producer generates messages periodically which are delivered to both consumers. We implemented the system in Python as well as WYVERN. In both implementations, the

producers and consumers are connected together by an event connector that's implemented using the RabbitMQ library. We changed both implementations to use an event connector implemented using Apache Kafka instead. Again, in the WYVERN implementation, we only had to change the architecture description, whereas in the Python implementation, we had to make widespread changes in the codebase.

Data Access

For this category, we changed the connectors for interacting with a MySQL database in a PHP-based web application to those for interacting with a MongoDB database. We implemented this scenario in our SQL to NoSQL migration case study. Details of this scenario are given in section 5.1.2.

Linkage

As explained in section 5.2.1, linkage connectors are used for binding names at compile time, not for enabling components to interact at run time. So, the connector abstraction mechanism in WYVERN doesn't support the implementation of linkage connectors. As a result, we haven't implemented any connector evolution scenarios involving linkage connectors.

Stream

For this category, we changed the RabbitMQ-based pipe connector in a system to an Apache Kafka-based pipe connector. We implemented a pipe-and-filter system with two filters that interact using a pipe connector. One of the filters generates random integers which are sent to the other filter via the pipe connector. We implemented the system in Python as well as in WYVERN. Initially, we implemented the pipe connector in both implementations using the RabbitMQ library. We then changed both implementations to use a pipe connector implemented using Apache Kafka instead. To do this modification, we had to make widespread code changes in the Python implementation. In the WYVERN implementation, however, the changes were restricted to the architecture description.

Arbitrator

For this category, we implemented a client-server system in which the client makes synchronous gRPC calls to the server through a circuit breaker. We implemented the system in Java and WYVERN. In the initial implementation, the circuit breaker is implemented using the Netflix Hystrix library. We changed the system to use the resilience4j library instead for the implementation of the circuit breaker. Developers today are interested in this scenario because Netflix has stopped active development of the Hystrix library. So, many existing codebases using the Netflix Hystrix library need to be migrated to the resilience4j library. Again, in the WYVERN implementation, we only had to change the architecture description, while in the Java implementation, we had to make widespread changes in the codebase.

Table 5.12: Connector evolution scenarios implemented using our connector evolution approach

Type of connector	Connector evolution scenarios implemented
Procedure call	1. ROS 1 service to ROS 2 service 2. ROS 1 action to ROS 2 action 3. ROS 1 service to synchronous gRPC connector 4. ROS 1 action to asynchronous gRPC connector
Event	1. ROS 1 topic to ROS 2 topic 2. RabbitMQ-based event connector to Apache Kafka-based event connector
Data access	Connector for MySQL databases to connector for MongoDB databases
Linkage	Not implemented because linkage connectors are not run-time entities
Stream	RabbitMQ-based pipe connector to Apache Kafka-based pipe connector
Arbitrator	gRPC connector with Netflix Hystrix-based circuit breaker to one with resilience4j-based circuit breaker
Adaptor	Not implemented as real-world scenario couldn't be found
Distributor	Not implemented because distributor connectors are low-level connectors

Adaptor

We couldn't find any real-world connector evolution scenario involving adaptor connectors. So, we haven't implemented any.

Distributor

Since distributor connectors are low-level connectors that never exist independently, we haven't implemented any distributor connector using the connector abstraction mechanism in WYVERN. So, we haven't implemented any connector evolution scenarios involving distributor connectors.

Summary

We have demonstrated the generality of our connector evolution approach by showing how it can be used to change connectors in a wide range of real-world connector evolution scenarios. Table 5.12 concisely shows the evolution scenarios we have implemented involving connectors from each category in Mehta et al.'s taxonomy of connectors.

5.3 Claim 3: Usefulness of Connector Compatibility Analysis in Prevention of Errors

In current practice, architecturally relevant information about connectors is often included in code. For example, in systems implemented using the Robot Operating System 2 (ROS 2) framework, the message delivery policy can be configured to be either best effort or guaranteed delivery [7]. This is done in the code using an API function call. Previous studies have found that such architectural knowledge can be lost over time [45]. However, such semantic information about connectors might need to be considered for selecting suitable replacement connectors when one or

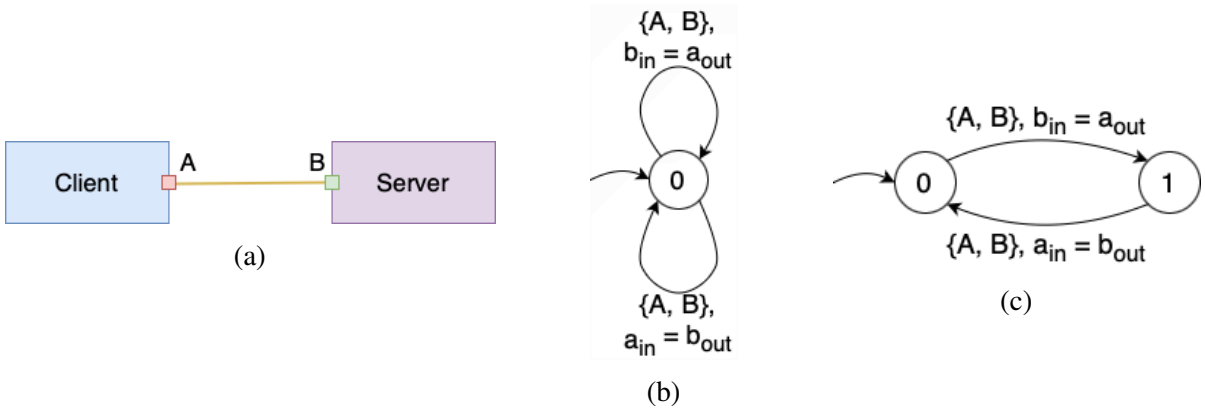


Figure 5.4: Constraint automata for (b) asynchronous and (c) synchronous RPC connectors that may be used in a client/server system

more connectors in a system have to be changed. In particular, architects might have to determine if a particular desired behavior in a connector being replaced is preserved in the replacement connectors they're considering. If the behavior is not preserved, errors might be introduced into the system after the connector is changed.

The connector abstraction mechanism we have developed can be used to specify relevant semantics of connectors explicitly rather than leaving it in code. In chapter 4, we discussed how the data transfer semantics of a connector can be specified using constraint automata. This enables checking whether a connector being replaced and a candidate replacement connector are compatible with respect to the specified semantics. By checking compatibility between the original connector and candidate replacement connectors early in the connector migration process at the time of selecting a replacement connector, errors can be prevented from being introduced into the system. In section 4.2, we described our approach for checking compatibility between connectors in terms of their data transfer semantics that's been specified using constraint automata. We demonstrate the usefulness of our connector compatibility analysis approach in preventing errors by presenting a few examples of the class of incompatibilities that can be detected using our approach.

5.3.1 Call Synchronicity

Constraint automata can be used to specify whether a procedure call connector enables synchronous or asynchronous method invocation. Replacing a procedure call connector that supports asynchronous calls with one that supports synchronous calls might lead to incorrect system behavior as the performance of the system might be impacted after the connector change. Our compatibility checking algorithm can detect incompatibilities between procedure call connectors with different method invocation semantics.

Consider the client/server system shown in figure 5.4a. Suppose the client and the server interact using a remote procedure call (RPC) connector that provides asynchronous method invocation. The semantics of the connector may be specified using the constraint automaton shown in figure 5.4b. As can be seen in the figure, the connector allows the client to make a call

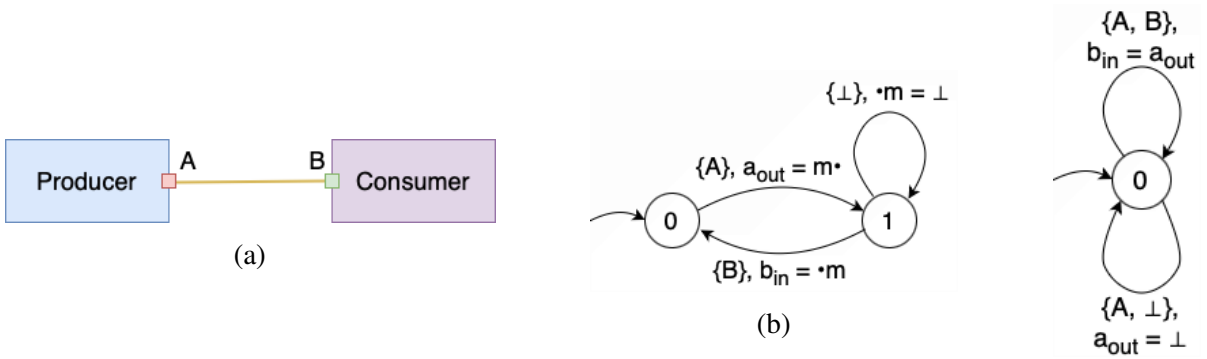


Figure 5.5: Constraint automata for event connectors with (b) guaranteed delivery policy and (c) best effort delivery policy that may be used in a producer/consumer system

while another call is in progress whose result has yet not been returned to the client. Suppose an RPC connector that supports synchronous method invocation is being considered as a candidate replacement connector. Figure 5.4c shows the constraint automaton that specifies the semantics of this connector. The candidate replacement connector doesn't allow calls to be made while a call is in progress. The connector starts in state 0. When a call is made, it transitions to state 1. In state 1, no further calls can be made. When the result is returned, the connector transitions back to state 0 after which another call can be made. The candidate replacement connector is incompatible with the original connector with respect to the specified data transfer semantics. Our compatibility analysis algorithm can detect this incompatibility before the change is actually made. Architects can use the results of the analysis to carefully consider if the candidate connector could still be used or if it should be rejected.

5.3.2 Delivery Policy

Our connector compatibility analysis approach can detect incompatibilities between connectors that use different delivery policies.

Consider the producer/consumer system shown in figure 5.5a. Suppose that the producer and consumer interact using an event connector. An event connector may provide guaranteed delivery of messages or it may provide only best effort delivery. Suppose that the event connector used in the system ensures guaranteed delivery. The constraint automaton for specifying the semantics of guaranteed delivery is shown in figure 5.5b. Here, we use a special reserved port, which we designate using $\perp$, to model any failure that may occur while sending a message. Any message that fails to reach its destination is modeled as being sent to port $\perp$. The connector starts in state 0. When a message is sent by the producer from port A, the message is stored in a buffer and a transition occurs to state 1. In state 1, the connector keeps trying to deliver the message to port B until it is successfully delivered. When the message is successfully delivered, the connector transitions to state 0.

Suppose a candidate event connector that only provides best effort delivery is being considered to be used as the replacement for the connector that supports guaranteed delivery. The constraint

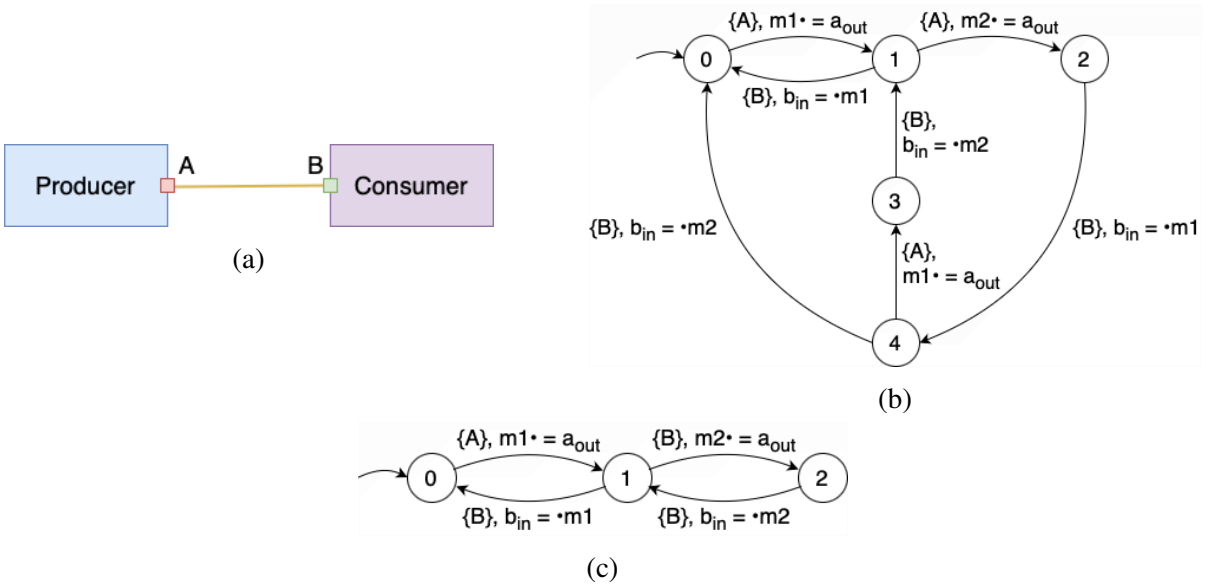


Figure 5.6: Constraint automata for buffered stream connectors with (b) oldest first delivery order and (c) most recent first delivery order that may be used in a producer/consumer system

automaton specifying the semantics for this candidate replacement connector is shown in figure 5.5c. As can be seen in the figure, when a message is sent from port A, it may be delivered successfully to port B or it may be lost. When there is a failure in sending a message, the connector doesn't try to resend it.

Replacing the connector that provides guaranteed delivery with the connector that only provides best effort delivery would lead to incorrect system behavior because messages may get lost. Our connector compatibility checking approach can detect this incompatibility between the connectors before the change is actually carried out, thereby preventing the introduction of errors in the system.

5.3.3 Message Delivery Order

Our connector compatibility analysis algorithm can be used to detect incompatibilities resulting from the order in which connectors deliver messages.

Consider the producer/consumer system shown in figure 5.6a. Suppose that the producer and the consumer interact using a buffered stream connector. For simplicity, we'll assume that the buffer is of size 2. The delivery order used for messages stored in the buffer may be oldest first (i.e., the message in the buffer that arrived first is delivered first) or most recent first (i.e., the message in the buffer that arrived last is delivered first) [67]. Suppose that the connector used in the system uses oldest first delivery order, while the candidate replacement connector uses most recent first delivery order.

Figures 5.6b and 5.6c show how the oldest first and most recent first delivery order semantics can be specified using constraint automata.

The constraint automaton for the connector that uses oldest first order is shown in figure 5.6b.

State 0 represents situations in which the buffer is empty. States 1 and 4 represent situations in which the buffer contains just one message. In state 1, the message is contained in memory cell $m1$ and in state 4, it is contained in memory cell $m2$. States 2 and 3 represent situations where the buffer is full. In state 2, the oldest messages is contained in memory cell $m1$ and in state 3, it is contained in memory cell $m2$. In each state, if a transition corresponding to message delivery is taken, then the oldest message is delivered in the transition.

Figure 5.6c shows the constraint automaton for the connector that uses most recent first order. State 0 represents situations in which the buffer is empty. Similarly, state 1 represents situations in which there is only one message in the buffer and state 2 represents situations where the buffer is full. In each state, if a message is delivered, the message in the buffer that arrived last is picked.

If the system is implemented by assuming oldest first delivery order, then replacing the connector used with the candidate connector that uses most recent first order would cause errors in the system. Our compatibility analysis algorithm can detect this incompatibility before the change is actually carried out and thus prevent the introduction of errors into the system.

5.3.4 Limitations of the Connector Compatibility Analysis Approach

Even though the constraint automata formalism that we have used for the specification of data relay semantics of connectors can be used to model connector semantics in a wide variety of situations, there are limitations to its expressive power. The formalism does not admit an explicit modeling of time. As such, events that are controlled by a timer cannot be modeled using the constraint automata formalism that we use. For example, a circuit breaker (see section 5.2.1) transitions from the open state to the half open state after a configured time period elapses. To model such time-related events, the formalism needs to be extended to incorporate clock variables.

Another limitation of our connector compatibility analysis approach is that when unrolling loops during the symbolic execution of a constraint automaton, we assume that values in one iteration do not depend on previous iterations. This may not always be true. Therefore, the symbolic execution algorithm needs to be extended to work without this assumption.

5.4 Summary

We evaluated the effectiveness of our connector evolution approach in making the task of connector evolution easy in two case studies. The results from the two case studies show that our approach makes the task of connector evolution much easier in cases where the port interfaces need not be changed. In these cases, the changes to be made are completely localized to the architecture description. We showed the expressiveness of our connector abstraction mechanism by implementing a wide range of connectors using the mechanism. Furthermore, we showed the generality of our connector evolution approach by using it to change connectors in a wide range of scenarios. Lastly, we demonstrated the usefulness of our connector compatibility analysis approach in preventing errors during connector evolution by presenting examples of the class of incompatibilities that can be detected using our approach.

Chapter 6

Conclusion and Future Work

In this thesis, we presented an architecture-centric connector evolution approach for making the task of connector evolution easier. Our approach involves integrating the architecture description of a system with its implementation when building the system. In our approach, we provide a mechanism for explicitly specifying the architecture of the system. The architecture specification describes the topology of the component connections and the type of connector used for each connection. Interfaces for components must also be specified in the architecture description. Components are then implemented such that they interact exclusively via these interfaces. In the component code, we prevent the direct use of libraries that can be used to implement connectors. We do this by the use of capabilities. By doing this, we prevent developers from bypassing the use of interfaces for component interaction. We provide an explicit abstraction mechanism for connectors. Based on the specification of component interfaces and component-connector attachments in the architecture description, the code that enables components to interact using a particular connector type can be generated using that connector's abstraction. Since components interact only through interfaces and the code implementing a connector is generated using its abstraction, references to a particular connector type are not spread across the codebase as happens in the current state of practice. Instead, they are localized to the architecture description. As a result, changing a connector becomes easy as only a few lines in the architecture specification have to be changed.

We evaluated the effectiveness of our approach in making the task of connector evolution easier in two case study scenarios. In the first case study, we migrated a suite of robotic software systems implemented using the first generation of the Robot Operating System (ROS) framework, i.e., ROS 1, to the second generation, i.e., ROS 2. In the second case study, we migrated a web application that uses a MySQL database, which is a SQL-based database, to use a MongoDB database, which is a document-oriented NoSQL database, instead. The two case studies demonstrate that our approach makes the task of connector evolution much easier in cases where the port interfaces need not be changed. In these cases, the changes to be made are completely localized to the architecture description.

We showed the expressiveness of our connector abstraction mechanism by implementing a wide range of connectors using the mechanism. We also showed the generality of our connector evolution approach by using it to change connectors in a wide range of scenarios.

In current practice, architecturally relevant information about connectors is often included in

code. Previous studies have found that such architectural knowledge can be lost over time. We have provided a mechanism in the connector abstraction mechanism we have developed to specify relevant semantics of connectors explicitly rather than leaving it in code. In this thesis, we focus on the data transfer semantics of connectors. We use constraint automata for specifying the data transfer semantics of a connector in our connector abstraction mechanism. This enables checking whether a connector being replaced and the replacement connector are compatible with respect to the specified semantics. Our compatibility checking algorithm is based on symbolic execution of the constraint automata for the connectors. By checking compatibility between the original connector and candidate replacement connectors early in the connector migration process at the time of selecting a replacement connector, errors can be prevented from being introduced into the system. We have demonstrated this by presenting a few examples of the class of incompatibilities that can be detected using our approach.

Future Work

The work presented in this thesis can be extended in several ways. Our architecture description enables the specification of only the component-and-connector view of a system. It would be useful to integrate a deployment view of the system as well. This would enable reasoning about the behavior of connectors in a system in different deployment environments. This would aid architects in deciding whether or not to change the connectors in a system. They might find, for example, that instead of changing the connectors, the quality attribute requirement(s) they are interested in might be achieved just by moving to a different deployment environment.

Several improvements can be made to our connector compatibility analysis approach as well. In our symbolic execution algorithm for a constraint automaton, when unrolling loops, we assume that the values in one iteration of a loop do not depend on previous iterations. The symbolic execution algorithm needs to be extended to work without this assumption. Alternatively, other approaches for checking compatibility, such as a bisimulation-based approach, may be developed.

In this work, we have simply assumed that the connector implementation code conforms to the data transfer semantics specified using a constraint automaton. A way to enforce the conformance of code to the specified semantics needs to be developed.

The tool support for connector compatibility analysis needs to be improved as well. For example, it might be easier for users to specify a constraint automaton using a graphical notation. The text-based specification of the automaton in our DSL should then be generated from the graphical specification.

Usability studies need to be conducted to understand how easy it is for practitioners to use both our connector evolution approach and connector compatibility analysis approach. Results from these studies can be used to guide further improvements in both approaches.

Bibliography

[1] Pipes and Filters pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters. Accessed: 2024-10-22.

[2] 2024 NoSQL Database Trend Report. https://ravendb.net/whitepapers/2024-nosql-database-trend-report. Accessed: 2024-10-09.

[3] Writing a Simple Action Client (Python). http://wiki.ros.org/actionlib_tutorials/Tutorials/Writing%20a%20Simple%20Action%20Client%20%28Python%29, . Accessed: 2024-10-22.

[4] Writing a Simple Action Server using the Execute Callback (Python). http://wiki.ros.org/actionlib_tutorials/Tutorials/Writing%20a%20Simple%20Action%20Server%20using%20the%20Execute%20Callback%20%28Python%29, . Accessed: 2024-10-22.

[5] Writing a Simple Service and Client (Python). http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29, . Accessed: 2024-10-22.

[6] Noetic Nimjemys: The Last Official ROS 1 Release. https://www.openrobotics.org/blog/2020/5/23/noetic-ninjemys-the-last-official-ros-1-release, 2020. Accessed: 2020-07-27.

[7] ROS 2 Documentation. https://docs.ros.org/en/foxy/index.html, 2024. Accessed: 2024-09-08.

[8] About ROS with Spot. https://support.bostondynamics.com/s/article/About-ROS-with-Spot-67882, 2024. Accessed: 2024-09-08.

[9] Spot SDK. https://dev.bostondynamics.com/docs/concepts/networking, 2024. Accessed: 2024-09-08.

[10] Rahma S. Al Mahruqi, Manar H. Alalfi, and Thomas R. Dean. A Semi-automated Framework for Migrating Web applications from SQL to Document Oriented NoSQL Database. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 44–53, 2019.

[11] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language Support for Connector Abstractions. In *European Conference on Object-Oriented Programming*, pages 74–102, 2003.

[12] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[13] Robert Allen, Rémi Douence, and David Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Fundamental Approaches to Software Engineering*, pages 21–37, 1998.

[14] Asil A. Almonaies, Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. A Framework for Migrating Web Applications to Web Services. In *Proceedings of the 13th International Conference on Web Engineering*, ICWE '13, pages 384–399, 2013.

[15] F Arbab and J. J. M. M. Rutten. A Coinductive Calculus of Component Connectors. In *Recent Trends in Algebraic Development Techniques*, pages 34–55, 2003.

[16] Dean N. Arden. Delayed-logic and finite-state machines. In *2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1961)*, pages 133–151, 1961.

[17] Uwe Aßmann and Andreas Ludwig. Introducing Connections Into Classes With Static Meta-Programming. In *Coordination Languages and Models*, pages 371–383, 1999.

[18] Uwe Aßmann, Thomas Genßler, and Holger Bär. Meta-programming Grey-box Connectors. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems*, pages 300–311, 2000.

[19] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, USA, 2008.

[20] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2): 75–113, 2006.

[21] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, 51(3), May 2018.

[22] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, and Daniel C. Sturman. A Case for Message Oriented Middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 1–17, 1999.

[23] Jeffrey M. Barnes, David Garlan, and Bradley Schmerl. Evolution styles: foundations and models for software architecture evolution. *Software & Systems Modeling*, 13(2):649–678, May 2014.

[24] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, USA, 2022.

[25] Thaís Batista, Christina Chavez, Alessandro Garcia, Cláudio Sant'Anna, Uirá Kulesza, Awais Rashid, and Fernando Castor Filho. Reflections on Architectural Connection: Seven Issues on Aspects and ADLs. In *Proceedings of the 2006 International Workshop on Early Aspects at ICSE*, pages 3–10, 2006.

[26] Don Batory. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1:355–398, 1992.

[27] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When Does a Refactoring Induce Bugs? An Empirical

Study. In *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 104–113, 2012.

[28] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[29] Lodewijk Bergmans and Mehmet Aksit. Composing Crosscutting Concerns Using Composition Filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[30] Michelle Berry. A Kafka Epic — How Checkr built a more reliable and performant background check platform by migrating to Apache Kafka. `https://engineering.checkr.com/a-kafka-epic-how-checkr-built-a-more-reliable-and-performant-background-check-platform-by-75f00e05c3ca`, 2019. Accessed: 2020-05-12.

[31] Robert L. Bocchino Jr., Timothy K. Canham, Garth J. Watney, Leonard J. Reder, and Jeffrey W. Levison. F Prime: An Open-Source Framework for Small-Scale Flight Software Systems. In *32nd Annual AIAA/USU Conferene on Small Satellites*, 2018.

[32] Grady Booch and Michael Vilot. The Design of the C++ Booch Components. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 1—11, 1990.

[33] Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly, USA, 2020.

[34] Yérom-David Bromberg and Valérie Issarny. Service Discovery Protocol Interoperability in the Mobile Environment. In *Proceedings of the 4th International Conference on Software Engineering and Middleware*, pages 64–77, 2004.

[35] Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4): 481–494, October 1964.

[36] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, September/October 2005.

[37] Greg Burd. Nosql. *;login: the USENIX Association newsletter*, 36(5), October 2011.

[38] Steve Cousins. Exponential Growth of ROS. *IEEE Robotics & Automation Magazine*, 18 (1):19–20, 2011.

[39] Patrick Cousot and Radhia Cousot. Semantic Analysis of Communicating Sequential Processes (Shortened Version). In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 119–133, 1980.

[40] Eric M. Dashofy, Nenad Medvidovic, and Richard N. Taylor. Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 3–12, 1999.

[41] Bernhard Dieber, Benjamin Breiling, Sebastian Taurer, Severin Kacianka, Stefan Rass,

and Peter Schartner. Security for the Robot Operating System. *Robotics and Autonomous Systems*, 98:192–203, 2017.

[42] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and Using Pluggable Type-Checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, 2011.

[43] Leticia Duboc, David S. Rosenblum, and Tony Wicks. A Framework for Characterization and Analysis of Software System Scalability. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 375–384, 2007.

[44] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified Protection Model of the seL4 Microkernel. In *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments*, pages 99–114, 2008.

[45] Martin Feilkas, Daniel Ratiu, and Elmar Jurgens. The Loss of Architectural Knowledge during System Evolution: An Industrial Case Study. In *IEEE 17th International Conference on Program Comprehension*, pages 188–197, 2009.

[46] David Garlan and Mary Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. December 1993.

[47] David Garlan, Robert Monroe, and David Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, November 1997.

[48] Michael W. Godfrey and Daniel M. German. On the evolution of Lehman's Laws. *Journal of Software: Evolution and Process*, 26(7):613–619, 2014.

[49] Katerina Goseva-Popstojanova, Aditya P. Mathur, and Kishore S. Trivedi. Comparison of Architecture-Based Software Reliability Models. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 22–31, 2001.

[50] Lars Grunske. Formalizing Architectural Refactorings as Graph Transformation Systems. In *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Network*, pages 324–329, 2005.

[51] Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesus M. Gonzalez-Barahona. The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review. *ACM Comput. Surv.*, 46(2), December 2013.

[52] Dan Hirsch, Sebastián Uchitel, and Daniel Yankelevich. Towards a Periodic Table of Connectors. In *Coordination Languages and Models*, pages 418–418, 1999. ISBN 978-3-540-48919-1.

[53] Igor Ivkovic and Kostas Kontogiannis. A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations. In *Conference on Software Maintenance and Reengineering*, pages 135–144, 2006.

[54] S.-S.T.Q. Jongmans, T. Kappé, and F. Arbab. Constraint automata with memory cells and their composition. *Science of Computer Programming*, 146:50–86, 2017.

[55] Ackbar Joolia, Thais Batista, Geoff Coulson, and Antonio Tadeu A. Gomes. Mapping

ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 131–140, 2005.

[56] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, and Jeromy Carriere. The Architecture Tradeoff Analysis Method. In *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems*, pages 68–78, 1998.

[57] Rick Kazman, Mario Barbacci, Mark Klein, S. Jeromy Carrière, and Steven G. Woods. Experience with Performing Architecture Tradeoff Analysis. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, page 54–63, 1999.

[58] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[59] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[60] Heiko Koziolek, Bastian Schlich, and Carlos Bilich. A Large-Scale Industrial Case Study on Architecture-based Software Reliability Analysis. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 279–288, 2010.

[61] Evandro Miguel Kuszera, Leticia Mara Peres, and Marcos Didonet Del Fabro. Exploring data structure alternatives in the RDB to NoSQL document store conversion process. *Information Systems*, 105, 2022.

[62] M. M. Lehman. Programs, Cities, Students — Limits to Growth? In *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, pages 42–69. 1978.

[63] Sebastian Lehrig and Steffen Becker. The CloudScale Method for Software Scalability, Elasticity, and Efficiency Engineering: a Tutorial. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, page 329–331, 2015.

[64] Bennet P. Lientz and E. Burton Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley Publishing Company, USA, 1980.

[65] Yannis Lilis and Anthony Savidis. A Survey of Metaprogramming Languages. *ACM Computing Surveys*, 52(6), October 2019.

[66] Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. *SIGSOFT Software Engineering Notes*, 21(6):3–14, October 1996.

[67] Nazanin Magharei, Reza Rejaie, Ivica Rimac, Volker Hilt, and Markus Hofmann. ISP-Friendly Live P2P Streaming. *IEEE/ACM Transactions on Networking*, 22(1):244–256, 2014.

[68] Dennis Mancl. Refactoring for Software Migration. *IEEE Communications Magazine*, 39 (10):88–93, October 2001.

[69] Jarrod McClean, Christopher Stull, Charles Farrar, and David Mascareñas. A preliminary cyber-physical security assessment of the Robot Operating System (ROS). In *Proc. SPIE*

8741, Unmanned Systems Technology XV, pages 341–348, May 2013.

[70] Nenad Medvidovic. On the Role of Middleware in Architecture-Based Software Development. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 299–306, 2002.

[71] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[72] Nenad Medvidovic, Peyman Oreizy, and Richard N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability*, SSR '97, page 190–198, 1997.

[73] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 44–53, 1999.

[74] Nikunj R. Mehta and Nenad Medvidovic. Understanding Software Connector Compatibilites Using A Connector Taxonomy. In *Proceedings of the First Workshop on Software Design and Architecture*, 2002.

[75] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 178–187, 2000.

[76] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. A Capability-Based Module System for Authority Control. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, pages 20:1–20:27, 2017.

[77] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A Security-Oriented Subset of Java. In *Network and Distributed Systems Symposium*. Internet Society, 2010.

[78] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript, 2008. Unpublished.

[79] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[80] Parastoo Mohagheghi and Reidar Conradi. An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 7–16, 2004.

[81] Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. The Pragmatic Programmers, LLC, USA, 2018.

[82] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely Composable Type-Specific Languages. In *ECOOP 2014 – Object-Oriented Programming*, pages 105–130, 2014.

[83] Flavio Oquendo, Brian Warboys, Ron Morrison, Régis Dindeleux, Ferdinando Gallo, Hubert Garavel, and Carmen Occhipinti. ArchWare: Architecting Evolvable Software. In *European Workshop on Software Architecture*, pages 257–271, 2004.

[84] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering*, pages 177–186, 1998.

[85] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. An Exploratory Study on the Relationship between Changes and Refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension*, pages 176–185, 2017.

[86] Ilian Pashov, Matthias Riebisch, and Ilka Philippow. Supporting Architectural Restructuring by Analyzing Feature Models. In *Conference on Software Maintenance and Reengineering*, pages 25–34, 2004.

[87] Erik Persson. *Shadows of Cavernous Shades: Charting the Chiaroscuro of Realistic Computing*. PhD thesis, Department of Computer Science, Lund University, 2003.

[88] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[89] Don Roberts, John Brant, and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[90] Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma. Refactoring for Software Architecture Smells. In *Proceedings of the 1st International Workshop on Software Refactoring*, pages 1–4, 2016.

[91] Saurabh Sarkar and Chris Parnin. Characterizing and Predicting Mental Fatigue during Programming Tasks. In *2017 IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering*, pages 32–37, 2017.

[92] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and Extensible Renaming for Java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pages 277–294, 2008.

[93] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. Correct Refactoring of Concurrent Java Code. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, pages 225–249, 2010.

[94] Frederik Schmidt, Stephen G. MacDonell, and Andrew M. Connor. An Automatic Architecture Reconstruction and Refactoring Framework. In *Software Engineering Research, Management and Applications 2011*, pages 95–111. 2012.

[95] Mary Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Studies of Software Design*, pages 17–32, 1996.

[96] Mary Shaw and Paul Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of the 21st International Computer Software and Applications Conference*, pages 6–13, 1997.

[97] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., USA, 1996.

[98] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and implementations for architectural connections. In *Third International Conference on Configurable Distributed*

Systems, pages 2–10, 1996.

[99] Bridget Spitznagel and David Garlan. A Compositional Approach for Constructing Connectors. In *The Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, Amsterdam, The Netherlands, August 2001.

[100] Bridget Spitznagel and David Garlan. A Compositional Formalization of Connector Wrappers. In *The 2003 International Conference on Software Engineering (ICSE'03)*, 2003.

[101] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, and Jason E. Robbins. A Component- and Message-Based Architectural Style for GUI Software. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 295–304, 1995.

[102] Christopher S. Timperley, Tobias Dürschmid, Bradley Schmerl, David Garlan, and Claire Le Goues. ROSDiscover: Statically Detecting Run-Time Architecture Misconfigurations in Robotics Systems. In *19th International Conference on Software Architecture (ICSA)*, pages 112–123, 2022.

[103] Mads Tofte. Essentials of Standard ML Modules. In *Advanced Functional Programming*, pages 208–229, 1996.

[104] Byron J. Williams and Jeffrey C. Carver. Characterizing software architecture changes: A systematic review. *Information and Software Technology*, 52(1):31–51, 2010.

[105] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for Reentrancy. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 173–182, 2009.

[106] Tianyin Xu and Yuanyuan Zhou. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Comput. Surv.*, 47(4), July 2015.

[107] Huaxi Zhang, Christelle Urtado, and Sylvain Vauttier. Connector-driven process for the gradual evolution of component-based software. In *2009 Australian Software Engineering Conference*, pages 246–255, 2009.

[108] Olaf Zimmermann. Architectural Refactoring: A Task-Centric View on Software Evolution. *IEEE Software*, 32(2):26–29, 2015.