Compositional Transformation of Software Connectors

Bridget Spitznagel

May 2004

CMU-CS-04-128

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Thesis Committee:
David Garlan, Chair
Phil Koopman
Mary Shaw
Martin Griss, UC Santa Cruz

This research was sponsored by the Defense Advanced Research Projects Agency and US Air Force Research Laboratory under grants F30602-00-2-0616, F30602-97-2-0031, and F33615-93-1-1330; by the National Science Foundation under grant CCR-0113810; by the National Aeronautics and Space Administration (NASA) under cooperative agreement NCC-2-1241; by the High Dependability Computing Program from NASA Ames, cooperative agreement NCC-2-1298; and by fellowships from the National Physical Science Consortium and Xerox. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of these organizations or the U.S. Government.

Keywords: Software architecture, software connectors, formal specifications, software engineering tools.

Abstract

Increasingly, complex software systems are being constructed as compositions of reusable software components. One critical issue for such constructions is the design and implementation of the interaction mechanisms, or connectors, that permit the various software components to work together properly. Complex systems also often have extra-functional requirements which these connectors must satisfy.

It is not always possible to find an existing connector that satisfies the requirements of the system, and at present it is time-consuming and difficult to create new kinds of connectors. A principled, compositional means of systematically constructing connectors is needed.

In this thesis I describe a new approach in which a basic connector type (such as RPC or data streams) can be augmented with selected adaptations or enhancements to produce a more complex connector type. I characterize a small set of structural transformations that can be applied compositionally to basic connector types to arrive at a wide variety of useful complex connector types. I give formal semantics for these transformations and demonstrate that it is possible to semi-automatically generate implementations of instances of the new connector types. I explore and evaluate this idea in the context of dependability, using transformations to increase the reliability of existing component interaction mechanisms.

Acknowledgments

"Oh, thank you! thank you! How can I express my gratitude?"

— Archibald Grosvenor in PATIENCE, Gilbert and Sullivan

First and foremost I would like to acknowledge my advisor David Garlan who has educated, guided, and encouraged me since my arrival at Carnegie Mellon. Thank you! thank you!

For their invaluable suggestions and timely advice, I am deeply indebted to Mary Shaw, Phil Koopman, and Martin Griss, my committee members; I would also like to thank Jeannette Wing and Stephen Brookes for their helpful comments on portions of this research.

It would be invidious to mention only a few of the many friends who have continually supported me. "My friends, I thank you all, from my heart, for your kindly wishes."[1]

Finally, I am inexpressibly grateful to my husband David Apfelbaum and our son Sean, for their love and especially for their *patience.*

[1] Frederick in THE PIRATES OF PENZANCE, Gilbert and Sullivan.

Contents

Chapter 1

Introduction

1.1 Motivation

Increasingly, complex software systems are being constructed as compositions of reusable software components. These components are often written independently and connected using glue code, which may have been produced by a third party. For example, in a three tier client-server system, the server, the database, and the code enabling them to communicate may have been acquired separately. The trend toward compositional software is a natural consequence of industry interest in the ability to incorporate third-party software and to reuse code.

One critical issue for such constructions is the design and implementation of the communication-based interaction mechanisms, or connectors, that permit the software components to work together. Architecturally, a connector is a discrete design element, representing a set of mechanisms that mediate interactions between components (more specifically, in this work I address connectors that enable *communication* between components); at the implementation level, however, the realization of such a connector is complex and consists of many parts. (For a more detailed definition, see section 3.2.) Getting connectors right is critical because otherwise the system may malfunction in subtle ways, may not give adequate performance, or may not work at all. Consider a system composed of separately developed merchants, customers, and a virtual bank, using a protocol for untraceable digital cash. A design flaw in the connectors might allow denial of service attacks, theft by a customer or merchant, violation of customer privacy, double-charging of customers, or even deadlock. Alternately, the system might function correctly, but be useless due to a sluggish response time. Connector design can have a significant impact on the system as a whole.

The communication in this example, a digital cash system, is not simple. The interaction mechanisms that support the communication will not be simple either: a simple connector, such as those provided as primitives in programming languages (e.g. procedure call), is not sufficient by itself, though the interaction mechanism may be built up from these primitives. The complex communication needs of the system can only be met by a semantically rich connector.

In addition to requirements related to the system's function, the connectors in software systems such as this example typically also have nontrivial extrafunctional requirements. These extrafunctional requirements contribute further to the complexity of the connectors. In the digital cash system, there will almost certainly be system-level requirements related to performance, dependability, and security concerns. These requirements affect both the overall system design and the connector design. To satisfy security requirements, a connector may have to support authorization and encryption. If a component is replicated for greater availability, the connector may be responsible for coordination; if the transport layer is lossy, the connector may be responsible for providing a stronger set of delivery guarantees. There may also be an upper bound placed on the latency of requests made via the connector. Further, these requirements are not orthogonal and generally interfere with one another. For example, coordination of replicated components may affect performance. Because of the need to satisfy these diverse, interacting requirements, the connectors will be even more complex than is implied by their basic function.

1.1.1 Motivating Example

Consider the following scenario, which will be revisited in Chapter 3. In this example, a remote procedure call connector C provides communication between a client component A and a server component B. The client sends a request to the server and expects to receive a response. Requests are not necessarily idempotent, but rather affect the state of the server, so that the same request may provoke a different response depending on the recent communication history. The server is subject to silent failure: it may crash (either between requests or while a processing a request), after which it will not respond until a human intervenes. If the server does not respond to a request within an interval t, the connector will report a timeout to the client. The communication channel, too, is not completely reliable, and a timeout (effectively the loss of a request or a response) may occasionally be reported to the client in place of a response even when the server is still operational.

In this example, we wish to *increase the dependability* of the system by adding two domain-specific enhancements to this connector C. With the first enhancement, rather than sending a timeout to the client, the connector will attempt to re-send the request that timed out; due to possible non-idempotency, duplicate detection and elimination will also be necessary to ensure that each request arrives at most once (consider the case where the response, rather than the request, was lost). With the second enhancement, if the attempts to re-send the request continue to fail, the connector will divert communication intended for the failed server to a "warm" backup server. But how might one create this enhanced connector?

1.2 Problems with State of the Art

Unfortunately, it is at present difficult to create the semantically rich connectors that these software systems need. There are two commonly available alternatives: to use an existing connector, or to write a connector from scratch. Neither alternative is adequate. On one hand, it is not always possible to find an existing connector that meets the needs of the system. As noted above, simple connectors are not sufficient by themselves. More complex middleware, such as CORBA, might meet the functional needs but prove inadequate for extrafunctional reasons such as real-time performance requirements. When no existing connector will satisfy the system's needs, using an existing connector is not a viable option.

On the other hand, creating a new connector is a difficult undertaking. Low-level knowledge of operating systems and auxiliary mechanisms such as stub generators is often required. In tension with this need for highly knowledgeable programmers (gurus) is the drive towards enabling less experienced, non-guru programmers to build systems from existing components. Since existing connectors will not always be adequate, these non-gurus will sometimes be faced with the task of creating new connectors in order to compose systems; while they may excel at the level of component authoring and component integration, it should not be expected that they would have the particular skills required for low-level modifications to communication mechanisms.

1.3 Partial Solutions

One possible solution is to require all new connector implementations to have a much greater degree of flexibility or modularity than has been common in the past. (Existing approaches include a predefined set of small composable modules available within a specific connector implementation [24] and the use of communication class libraries that can be specialized through the use of object-oriented inheritance [65].) This approach naturally offers little or no benefit to any established code base that uses currently existing connector implementations. Furthermore, unless these are made sufficiently open-ended, it places a burden on the connector implementors of foreseeing (and providing for) all future modification needs; if they fail to foresee a particular kind of extrafunctional requirement, then this solution offers no benefit to a system architect who faces that particular requirement. More-flexible connector implementations may be desirable in their own right but are not an adequate solution.

A different (and far less principled) partial solution that is currently employed is to make ad-hoc modifications to particular systems by hand. If, for example, a connector between component A and component B sometimes needs to compress the communicated data "on the fly" to reduce latency, additional code to compress and uncompress the data might be added to the implementations of A and B, or (depending on the connector implementation) to their automatically generated "stubs", to achieve this. Altering either the stubs or the connector-provided facility that generates them will require a comparatively deep understanding of the connector implementation; modifying A and B is a more likely approach, particularly if time is short. Either approach will increase the difficulty of maintaining the system as communication concerns become entangled with the component services, and making additional ad-hoc modifications to the communication becomes increasingly difficult to "get right" in combination with the modifications already in place.

1.4 A New Approach

What is needed is a principled way to produce new connectors systematically and at low cost. This is possible if we construct new connector variants compositionally: in this approach, basic connectors are augmented with a collection of selected adaptations or enhancements to produce a more complex connector that meets the requirements. For example, an existing inadequate connector might be augmented so that it will satisfy availability and security requirements, with acceptable effects on its performance.

This construction of new connectors requires a set of compositional operators, which I will refer to as *connector transformations.* A transformation can be thought of as a function that takes a connector as one of its arguments, and produces a new, enhanced connector. In order to avoid some of the pitfalls observed in section 1.3 associated with the alternative of writing a flexible implementation of a particular connector type, it must be possible to produce a *wide variety* of complex connectors both in terms of variety in the type of connector that can be enhanced and in terms of the domain-specific enhancements that can be performed. In order to support variety in enhancements while still keeping the number of transformations manageable, in my approach these transformation operators are general-purpose, rather than domain-specific, and are compositional, so that results more complex and varied than any single transformation are possible.

The idea of composing connectors compositionally is a general idea. For my thesis research I have explored this idea in the more limited context of an appropriate domain, *dependability.* I argue that this compositional approach is useful when constructing connectors that are required to have extrafunctional properties, and that one example of such an area is the domain of dependability, where this technique can be used in hardening connectors; system integrators can select from a set of well-known and commonly-used enhancements to make connectors robust against a variety of threats.

1.4.1 Examples of Transformations

In the example posed earlier, two dependability enhancements were suggested. As we will see in Chapter 3, each of these enhancements can be constructed from connector transformations. Let's briefly consider two transformations that would be used for the first enhancement, resending unacknowledged requests. To record transmissions and later replay them, I use a *sessionize* transformation, which adds state to create a communication session. To tag these transmissions with a unique identifier so that duplicates can be detected at the receiving end, I use a *data translation* transformation, which modifies the data being communicated. Other transformations include those that add a new party to the communication (such as the additional server in the second hypothetical enhancement), and those that combine two connectors.

1.4.2 Thesis Statement

This thesis demonstrates that

> *We can define a small set of basic transformations that, when applied compositionally to simple communications-based connectors, produce a wide variety of useful complex connectors. These transformations can also be given a formal interpretation that allows us to understand their properties with respect to communication protocols. Furthermore, we can build a transformation-based tool that facilitates the generation of implementations of these new connectors.*

For the thesis research I have explored this idea in the context of a specific domain, *dependability*, showing how connector transformations can be used to capture common techniques of that domain.

1.4.3 Elaboration of Claims

The thesis statement is composed of three separate claims.

First, "we can define a small set of basic transformations that, when applied compositionally to simple communications-based connectors, produce a wide variety of useful complex connectors." That is, a small set of basic transformations can account for a wide range of practical communications-based connectors. To keep the set small and the transformations basic, but still enable a wide variety of enhancements, the transformations must be non-domain-specific. It must then be possible to instantiate these transformations so that they can be applied as domain-specific enhancements. The transformations are to be applied to communications-based connectors, beginning with a base set of simple connectors; transformations may be again applied to the resulting connectors, building up a more complex connector step by step. The resulting connectors must be useful: the base connectors should be ones that are in use today, the domain-specific enhancements demonstrated should be ones that are commonly accepted within that domain as useful, and the connector implementations that are generated (by the tool of the third claim) should have acceptable performance characteristics, which I will return to in a moment. A wide variety of enhancements includes cross-domain (that enhancements are possible in more than one domain) and intra-domain (that a large fraction of connector-related well-known enhancements within a specific domain can be described in terms of connector transformations) coverage.

Second, "these transformations can also be given a formal interpretation that allows us to understand their properties with respect to communication protocols." I assert that it's possible to provide a formal characterization with respect to these transformations that can be used to perform analyses that answer important questions about the soundness, compositionality, and transparency of a particular connector transformation. Some pairs of transformations are not commutative with respect to one another, and not all combinations of transformations are necessarily reasonable; an inappropriate application of transformations could result in an unsound connector that is subject to deadlock, or in a connector that obviously does not have a desired new property (such as increased reliability that ensures that errors of some particular kind will be masked). Sometimes a pair of transformations may be applied in either order with different but non-catastrophic effects; it is useful in this case to be able to determine that the transformations are non-commutative, as a first step in deciding which ordering is most desirable for a particular goal. It is also desirable to be able to determine whether the protocol of a complex connector remains compatible with the interfaces of the components its simple predecessor connected (that is, to the components the change appears transparent with respect to protocol), or whether the components must be altered or an additional, mismatch-resolving, transformation must be applied to the complex connector to return it to compatibility.

Third, "we can build a transformation-based tool that facilitates the generation of implementations of these new connectors." The compositional approach allows rapid, easy development of complex connectors that are "good enough" for many practical systems. Support for at least three connectors, and at least three transformations each, is necessary to demonstrate breadth. In section 7.5.1 I characterize acceptable performance overhead in terms of the communication latency experienced by the original connector (in short, if overhead is roughly the same as the original same-host latency for the connector, then it will be insignificant compared to intranet or internet latency and thus likely to be acceptable for many distributed applications). Connectors with better performance characteristics could probably be written by hand, and should be in cases where that is important enough to justify the expense.

The compositional approach provides a principled *middle ground* (which, I argue above, does not currently exist) between expensive connectors hand-tailored to their requirements and off-the-shelf connectors that may not provide a good fit.

1.4.4 Summary of Contributions

The primary contribution of the thesis is the development and demonstration of a new engineering basis for component interactions that reduces the development cost of domain-specific connector enhancements by separating a monolithic modification into a set of smaller transformations. This relates to the first claim of the thesis statement (that we can use compositions of transformations to produce useful complex connectors).

This thesis also demonstrates a new variety of generation technology, in which a diverse range of new connector implementations can be semi-automatically generated from a small set of base connectors by applying domain-specific instantiations of connector transformations. This relates to the third claim of the thesis statement (that we can build a tool to apply transformations).

A further contribution is the clarification of the nature of connection. The space of software connectors appears at first to be complex and unordered, but it can be reduced to a more ordered and simpler space by considering complex connectors in terms of compositions of connector transformations and simpler connectors. This falls out of the first claim of the thesis.

Finally, in the area of applied formal methods, this thesis presents a formalism that is targeted at software engineers; it supports analyses that are intended to address engineering concerns. This corresponds to the second claim of the thesis.

1.5 Plan of Dissertation

Chapter 2 describes related work and introduces foundational work in software architecture on which this thesis builds, including the idea of a software connector as a "first class citizen."

Chapter 3 defines in more depth the concepts referred to in this thesis, such as “connectors” and “connector transformations.” A categorization of generic[1] connector transformations is introduced. In Chapter 4, a number of common dependability-enhancing modifications are described in terms of compositions of such connector transformations.

In Chapter 5, I give a formal basis for connector transformations, describing communication-based connectors as structured protocols and connector transformations as transformations on these protocols. The chapter covers an introduction to the syntax, the structure of a connector and a connector transformation in this formalism, and an extended example that leads into a catalog of patterns for describing connector transformations in terms of their effects on connector protocols.

The tool support for generating implementations of enhanced connectors is described in Chapter 6. I discuss the attributes that such a tool should provide, describe the specific approach taken for the particular tool that I have implemented, and indicate strengths and limitations of this approach. I give a simple example illustrating what a user of this tool must do to generate a new connector, and I give an overview of the specific base connectors and types of connector transformations that are currently supported by this tool.

Chapter 7 evaluates the claims of the thesis statement. I describe the results of case studies that I have undertaken, give a comparative task analysis of the connector transformation approach as compared to an “ad-hoc modification” approach, and describe the outcome of two informal experiments, one to determine the degree of performance overhead introduced by automatically generated connector transformation implementations and one to determine how long it takes a novice user of the tool to carry out a modification to a connector.

In Chapter 8 I describe design decisions, tradeoffs, and interesting issues that arose in the preceding chapters, and I discuss some intrinsically hard problems whose solutions are out of the scope of this thesis. Chapter 9 gives a summary of the thesis contributions and of future work in this area.

[1]We will use the word “generic” as shorthand for the unwieldy phrase “not specific to an application domain.”

Chapter 2

Related Work

2.1 Introduction

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. Additional related areas are protocol specification and analysis; generation of implementations; and design patterns. The validation of this work makes use of a survey of common dependability-enhancing techniques ([35] gives models and techniques for improving the reliability of software systems) as well as several connector implementations and existing software systems that use them. In this chapter I describe the relationship of my work to each of these areas.

2.2 Software Architecture

Software architecture deals with the description of a software system abstractly in terms of *components* and *connectors*; my work focuses on facilitating the construction of enhanced kinds of connectors.

Shaw et al. [51] 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 (in my work, I specifically address connectors that enable *communication* between components). 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 are depicted for reference here in section 3.2).

The treatment of connectors as first-class entities [48] 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. In this section I will touch specifically on: the existing support for connectors in several ADLs, particularly in relation to compositional construction or combination of connectors; existing means of classifying connector types, which relate to my generalization of domain-specific modifications into a set of "generic" connector transformations; and communication-related techniques for resolving component mismatch, which relate to mismatch-resolving connector transformations.

2.2.1 Connectors

Here I survey some existing architectural description languages' means of modeling connectors, focusing on support for compositionally constructing new connector types or instances.

In C2 [61], 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. In this architectural style, a specific limited kind of composition of connectors is permissible; two connector instances can be directly connected so that one passes messages down to the other. This composition of instances, however, does not result in a new connector type.

The Acme [21] architecture description language allows hierarchical architectures in which a component or connector may have a representation as a subsystem itself composed of components and connectors. This means of decomposing connectors could be used to support the depiction of "complex" connectors.

UniCon [51], which addresses the need for connector implementations to affect multiple implementation artifacts, is discussed below (section 2.4.1).

One benefit of the treatment of connectors as first-class, as Allen demonstrated with Wright [3, 1], 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 [4], 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. The work in this thesis builds on some of the ideas of Wright (as we will see in Chapter 5). In Wright, a connector may be specified as a composition of subprocesses, but Wright has no general idea of higher-order operations to recombine existing subprocesses into a different connector.

The Reo [5, 6] 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 constructed 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 homogenous: 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. My work has a different focus: to provide a means of enhancing existing connector types and connector implementations, which may already be rather complex themselves.

At a higher level of abstraction, Medvidovic and Taylor's classification of existing architecture description languages [39], 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. In this work I strive to provide a means of creating, and generating implementations of, more complex connector types.

2.2.2 Taxonomy

My work also builds on, and potentially contributes to, the classification of connector types into a taxonomy. In Chapter 3, I present a set of "generic" connector transformations, which was created as a partial taxonomic generalization of domain-specific *modifications* to connectors.

The feature-based classification for architectural styles given by Shaw and Clements

[50] 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, such as the following.

A "periodic table"-inspired classification by Hirsch, Uchitel, and Yankelevich [25] proposes a set of properties (such as "knows target," "synchronous," and "one way") 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. The consideration of such a set of properties was useful in my selection of "different" connector types (section 6.3.4) for demonstration of support for generating implementations of connector transformation based variants of connector types: to say that two connector types are different in some interesting way requires such points of comparison.

Mehta, Medvidovic, and Phadke [40] 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 a taxonomy can be used as an aid in the synthesis of new varieties of connectors. This idea relates to my work in that I provide a means for modifying or enhancing extrafunctional characteristics (such as aspects of reliability); a taxonomic understanding of the potential interactions between specific non-orthogonal desired characteristics would provide a complementary form of assistance for a software connector designer's preliminary selection of characteristics to be thus enhanced.

The connector transformation approach offers a different kind of leverage to the connector classification effort; variant connector types could be described as a "basic" connector type (as identified by Shaw et al. or Mehta et al.) plus a collection of transformations.

2.2.3 Architectural Mismatch

My work shares common ground, in its motivation, with research in identifying and resolving communication-related forms of mismatch: just as the need to enhance extrafunctional characteristics (as mentioned above) is one motivation for people who modify connectors, the need to repair mismatch between two intended components of a system is another such motivation.

Shaw observes that when two mismatched components are unable to communicate via existing connectors, one option is to construct or modify a *connector* that will then operate to resolve the mismatch [49]. Two kinds of connector-related architectural mismatches (identified by Garlan, Allen, and Ockerbloom [20]) are incompatible data models and conflicting assumptions about the communication protocols. These mismatches correspond to two of the connector transformations that I have identified: one that can translate between data formats (section 3.4), and one that produces a connector that appears to "speak" different protocols at different interfaces (section 3.12).

Yellin and Strom [64] describe a means of semiautomatically producing *component adaptors* to overcome forms of protocol mismatch between components (one of the two kinds of mismatch just mentioned above). If, instead of considering the resulting system as "(components plus adaptors) plus connector," one describes the adaptors as part of a new connector type, "components plus (adaptors plus connector)," this approach bears a resemblance to, and could be used to implement, the kind of connector transformation I describe as a "splice" (section 3.12). However, *conceptually* my work treats a spliced connector somewhat differently, as being a combination of (parts of) two connectors A and B with an A-to-B adaptation between them.

Another technique for dealing with mismatch, Deline's Flexible Packaging [17], separates the component's functionality (ware) from its assumptions about the communication infrastructure (packaging); mismatches in packaging can then be overcome by replacing the ware's packaging with one that is a better match for the rest of the system. As with component adaptors, one might choose to view the "packaging" as actually being part of the connector. The Flexible Packaging approach is elegant, but requires system designers to use components that have been flexibly packaged, and in general such components are not currently available; thus, the approaches of modifying a connector (perhaps using the connector transformation approach) or writing component adaptors are an alternative better suited to the conditions that prevail today.

2.3 Protocols and Formal Notation

This thesis builds on process algebras such as CSP and CCS [26, 41]. In particular, I apply to FSP [36] some of the structure of Wright [3] in order to describe protocols of software connectors and to describe connector wrappers as transformations of these protocols. Wright's decomposition of connectors into interfaces (or *roles*) and interactions (or *glue*) enables explicit identification of the communicating parties and their obligations as well as compatibility checks. My work, however, goes beyond that of previously published work on Wright by further decomposing the connector and promoting reusability of "wrapper" interaction elements that represent connector transformations.

Building on the idea of first-class connectors, Lopes, Wermelinger, and Fiadeiro have investigated the notion of "higher-order" connectors as a formal framework in which to create connectors compositionally. They base their work on category theory and CommUnity, a Unity-like parallel program design language [34]. The categorical semantic basis of their work gives it a different (formally elegant but possibly less intuitive to system architects who lack experience in category theory) form of compositional framework than the formalism presented here in Chapter 5. In addition I am interested in analyses that support engineering concerns, in the hopes of producing a formalism that appears comparatively attractive to my target audience, software engineers who have a need to produce connector variant implementations; this goal gives my approach to formal description of connector transformations a less rigorous, more practice-oriented focus.

A different kind of composition of formal descriptions is employed in Larch [22], an approach to formal specification. In Larch, a module's specification is structured as two tiers. The top (interface) tier is specialized to a particular programming language and describes the module's observable behavior. The bottom tier is independent of that language and is used to describe mathematical abstractions that can be reused and recombined. This kind of composition is interesting but fundamentally different from the forms of composition I use here, which combine pieces of formal description that are (as compared to the two-tier structure of Larch) all on a single level. Larch also supports intra-tier composition. Within the language-independent tier, a *trait* (unit of specification) can *include* other traits, thus providing a means of decomposing a specification into smaller pieces; a trait may include another in order to further specialize it, add to it, or rename items in it, or it might include several in order to combine them. Similarly in the language-dependent tier, an interface specification can import other interface specifications.

In addition to compositionality of formal protocol specifications, my work is somewhat related to the concept of creating new *protocols* by composition of simpler protocols. Although network protocols are by nature concurrent, one may arrive at simple protocols (suitable for composition) by considering an equivalent sequential behavior [57]. Some properties of safety and (given certain restrictions) liveness can be predicted in an incremental way using a finite state machine model [55]. Ensemble [62] enables the construction of an adaptive protocol composed of stacked micro-protocol modules. The *x*-Kernel [44] project has also used micro-protocol composition to design and implement a dynamic architecture for flexible protocols that take advantage of operating system support for efficient layering. Conduits+ [28] also provides a framework for network protocol software, with a focus on reuse aided by design patterns; layered protocols are composed from conduits (software components with two distinct "sides") and information chunks (which flow through the conduits). Another example of protocol composition is the Fox Net [11] implementation of the TCP/IP protocol suite. In this approach to composition, each module implements a single protocol in the SML language in a manner that ensures safe composability of the protocols. It is them possible to build entire protocol stacks from these independent mix-and-match modules, such as a stack of TCP over IP over ethernet, or a stack of a TCP variant without checksums directly over ethernet. These illustrate points in the space of possible granularity of the pieces being composed, and the effect of the chosen granularity on the degree of variation possible from a set of modules.

2.4 Generating Implementations

Here I consider several topics that relate to my approach of compositionally generating implementations of variant connector types: first, generation and combination of existing connector instances; next, techniques for creating variants of *specific* connector types; and finally, other compositional forms of generation that might be applied to connectors.

2.4.1 Connector Implementations

UniCon [51] 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. In contrast, my focus is on the creation, from existing implementations of connector types, of implementations of new variants of these connector types.

Similarly, ArchShell [38], 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. The implementation generation in my work differs from these in its focus on modifying existing systems by augmenting existing connector instances to create implementations of new connector types.

In their exploration of the use of off-the-shelf middleware to implement C2-style connectors in a distributed software system, Dashofy, Medvidovic, and Taylor [16] briefly discuss the possible merits of combining two or more middleware implementations within a single "virtual connector" so that either implementation may be selected. This form of composition somewhat resembles the connector transformation that I have named *aggregate* (section 3.10).

2.4.2 Connector Variations

My work provides a means of generating connector variants. As observed in section 1.3, some means of generating connector variations already exist, but they have limitations. For a specific type of connector, such as RPC, work has been done (e.g. by Hiltunen and Schlichting [24] and Zelesko and Cheriton [65]) in delaying the binding of some design decisions, such as the level of reliability, to make the implemented connector more flexible and appropriate for a wider range of applications; the decisions are not bound until the connector is integrated into a system. One approach is to have a set of small modules [24]. The options for behavior are classified into categories, such as call semantics (synchronous or asynchronous), and communication semantics (degree of reliability); micro-protocol modules, selected from these categories, are composed as in decompositional protocol synthesis. Another approach is to use object-oriented inheritance to specialize communication class libraries [65]. My work differs from these in its focus on producing new connector types that may be based on a variety of existing connector types.

System-level support mechanisms, usually called *interceptors*, are available for implementations of some commonly used connectors. Interceptors facilitate the insertion of arbitrary application-level wrapper code. Such code may be used to enhance fault tolerance [42] and security [18] of COTS components, or to add instrumentation [29]. By their nature these efforts are specific to a connector implementation and/or set of system libraries. Where available, interceptors could be used to *facilitate* the introduction of some kinds of connector transformations. Conversely, connector transformations can be used to provide a more principled framework for thinking about and decomposing the kinds of modifications that interceptors are currently used to introduce.

2.4.3 Other Approaches to Generation

GenVoca [10] takes a domain-specific compositional approach and illustrates the leverage that can be gained from restriction to a particular domain. Connector transformations, in contrast, are generic rather than being domain-specific. Connector transformations are also smaller in granularity than the building blocks of GenVoca.

Booch Components [12, 13], 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. This addresses the problem of programmers who tend to rewrite their own versions of existing components on the grounds that, in abstract functionality, an existing class matches their needs but its implementation doesn't satisfy an extrafunctional requirement (such as performance or memory usage). My work is similar in its distinguishing the small number of generic connector transformations and the wide potential range of domain-specific instantiations of them, but does not enumerate a vast predefined set of these instantiations for transformation users to browse at length; rather they are created as needed via the use of code fragments, parameters, and other inputs to the connector transformation implementation generation tool.

Aspect-oriented programming [31] presents a different kind of compositionality. Orthogonal "aspects" or "cross-cutting concerns" of a system are expressed separately (and in different languages) and are subsequently woven together into a single program. In a more traditional approach, a cross-cutting concern would have been spread out in several implementation artifacts, making it more difficult to modify, understand, and maintain. My work has some similarities, which arise naturally from the desire to localize code that cuts across multiple implementation artifacts; however, I focus only on communication-related concerns and do not further distinguish the domain of a particular communication-related enhancement. In my approach to connector implementation generation, new pieces of code that represent desired connector transformations are woven in to the existing program at specific sites (associated with the existing connector instance's implementation), which will be described in section 6.2.3. These pieces of code are written in the language of the target system. Unlike a typical aspect-oriented approach, also, I am concerned particularly with the ability to compose multiple pieces of code (representing transformations) that address a single concern, communication.

2.5 Patterns

Another area of related work is design patterns. A design pattern is an application-independent design fragment that can be reused to solve a well-identified problem [19, 14]. These patterns have a structure composed of *participants* (to be filled in) that have a particular relationship to one another. For example, the problem of decoupling the producer of a piece of data from observers of its value is solved by the "Observer" pattern; the pattern allows one to change the number and type of data observers without changing the data producer. Connection transformations can be viewed as a class of pattern for adapting component interaction mechanisms; my work goes beyond pattern identification and codification to develop tools for applying the transformation.

Design patterns can be composed; many patterns complement one another and can, though they need not, be used together. When two patterns are composed, sometimes one can be viewed as filling in a specific "participant" within the structure of the other. In this sense the composition of connector transformations is similar to pattern composition. Chapter 5 describes transformations in relation to a connector's *roles* and *glue* (which are to be filled in, as with the participants in a pattern's structure), and when two connector transformations are composed, one of them can be viewed as part of the *glue* of the other.

My work also *uses* patterns: the implementation generation tool described in this thesis employs the *Proxy pattern* (from Gamma et al. [19]) to introduce transformations into the connector implementation. This approach is described in section 6.3.1.

2.6 Dependability

Avizienis, Laprie, and Randell [7] have defined four standard categories of techniques for achieving greater dependability in software: fault prevention, fault tolerance, fault removal, and fault forecasting. Lyu [35] has assembled an in-depth survey of well-known fault tolerance techniques, which I use in this thesis (particularly in Chapter 4 and Chapter 7) to demonstrate that a reasonable degree of coverage of the desirable augmentations within this particular domain that are relevant to a communication-based connector can be achieved with respect to their description in terms of compositions of connector transformations, and that a useful range of such augmentations can be implemented through the use of connector transformations. Chapter 4 of this thesis gives a thorough overview of these techniques (prior to describing their connector transformation based construction) which will not be repeated here; in particular, the list of techniques includes those based on recovery blocks (Randell and Xu [46], Kim [32]), N-version programming (Avizienis [8]), N self-checking programming (Laprie et al. [33]) as well as graceful degradation (Shelton et al. describe a scalable architecture-based approach [52] in which valid system configurations are enumerated through the use of input/output dependency based groupings of components into subsystems).

Work in dependable software architectures takes specific dependability techniques (often those involving multiple versions of a component) and incorporates these techniques at the architecture design level. Simplex [47] and Multi-Versioning Connectors [37] are examples of two different approaches that are based on the same concept, component redundancy, but focus on different architectural elements to achieve their results. In the Simplex approach, several versions of a component, differing in performance and reliability, are aggregated into a single apparent *component* to provide (ideally) the best features of each. The Multi-Versioning Connectors approach, on the other hand, is motivated by the need to seamlessly upgrade components and thus to be able to dynamically vary and compare the versions of a component that are in use. The M-V-C *connector* is responsible for coordinating a component's versions, concealing their existence from the rest of the system, and monitoring their performance to determine whether the upgrade is actually desirable; from the perspective of my work, it provides an interesting instance of a complex enhancement to a connector that may be constructed from smaller enhancements.

2.7 Case Studies

I use connector transformations to add the Kerberos [43] protocol (version 5) to a connector in Chapter 7 to demonstrate that connector transformations are applicable in a non-fault-tolerance context. I apply connector transformations to Sun's Java RMI [59], Sun's Java Message Service [58], and Simmons et al.'s Inter-Process Communication [53] connector implementations. I tested the use of connector transformations in the context of some existing software systems written by others: the Visualization for Algorithm Development (VisAD) Java library [23], and Sun's canonical blueprint application for Java 2 Enterprise Edition web-commerce (known as the J2EE Pet Store) [60].

Chapter 3

Connector Transformations

3.1 Introduction

The thesis statement presented in 1.4.2 asserts that "we can define a small set of basic transformations that, when applied compositionally to simple communications-based connectors, produce a wide variety of useful complex connectors." In this chapter I define such a set, describing each transformation and giving a variety of instances in which it might be used (subsequent chapters will further support the claim of a wide variety of results).

I first review software connector, define the term "connector transformation", and then introduce a collection of generic connector transformations that will be referred to in the remainder of this document. For each transformation I give examples of use.

To illustrate breadth within a domain, Chapter 4 gives a collection of commonly-used dependability augmentations and indicates how each may be constructed with one or more connector transformations.

3.1.1 Motivating Example

I return to the scenario introduced in Chapter 1 and sketch a solution based on connector transformations. The solution illustrates how transformations may be composed both to achieve a single complex augmentation by using more than one transformation and to achieve multiple augmentations in a single connector. This example will be used throughout the thesis; the specific transformations used here will be explained later in this chapter.

In this scenario, a remote procedure call connector C provides communication between a client component A and a server component B. The client sends a request to the server and expects to receive a response. Requests are not necessarily idempotent, but rather affect the state of the server, so that the same request may provoke a different response depending on the recent communication history. The server is subject to silent failure: it may crash (either between requests or while a processing a request), after which it will not respond until a human intervenes. If the server does not respond to a request within an interval t, the connector will report a timeout to the client. The communication channel, too, is not completely reliable, and a timeout (effectively the loss of a request or a response) may occasionally be reported to the client in place of a response even when the server is still operational.

In this example, we wish to increase the dependability of the system by replacing the original connector C with an enhanced connector C'. This connector has two independent enhancements (each of which is composed from connector transformations). First, rather than sending a timeout to the client, the connector attempts to re-send the request that timed out. Second, if the attempts to re-send the request continue to fail, the connector will divert communication intended for the failed server to a "warm" backup server.

For the first enhancement, because the requests are not idempotent and the reason for the timeout is not (yet) known, it is not enough to simply cache the request at the client side and re-send it; perhaps the server is still alive, and only the response was lost. We must also add a means of duplicate elimination, such as caching the response at the server side, and associating a unique id with each cached request (and corresponding response), so that a lost response can simply be resent rather than regenerated. As will be seen later, this enhancement will require both a *sessionize* transformation (to record and replay) and a *data translation* transformation (to add a unique id).

For the second enhancement, we will be adding a "backup server" that duplicates the functionality of the existing server. Since the server has state, it is necessary either to use a "warm" backup server (that is, one that has the same state as the primary server should), or to get the backup server caught up with recent happenings. Therefore, in addition to using an *add switch* transformation to add the backup server, we will also require another transformation. Either the backup server must function

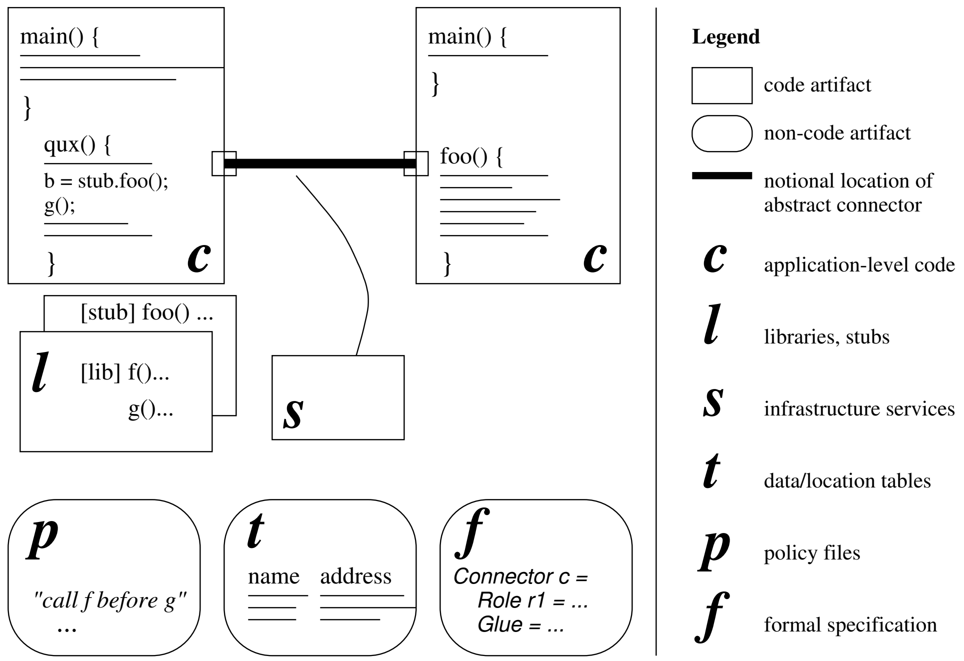

Figure 3.1: Concrete artifacts comprising a connector

as an *observer* while the primary is operational (essentially eavesdropping on all client requests), or the request history must be recorded with *sessionize* so that, after the failure of the primary server, sufficient history can be replayed to warm up a hitherto entirely dormant backup server.

3.2 Connectors

Before discussing connector transformations it is important to understand what is meant here by a connector, both when working at the level of an architectural abstraction (as it would be used at design time) and at the level of implementation (when it is actually a collection of several concrete artifacts).

Architecturally, a connector is a discrete design element, representing a set of mechanisms that mediate interactions between components. Shaw et al. [51] describe connectors, their realization at the implementation level in terms of a collection of dispersed and varied concrete artifacts, and different kinds of interactions that connectors may embody (such as communication, resource contention, and scheduling concerns). Specifically, in this work I address connectors that enable *communication* between components. A connector has some number of *roles* (connector interfaces); each role represents a participant in the communication. For example, consider a remote procedure call connector: it has two roles, one role for the participant that makes the call, and another role for the participant that receives the call. An architect attaches the roles of a connector to the *ports* (component interfaces) of a set of components. The roles define behavior with which these ports must be compatible; a role's specification indicates the minimum that is required of a component in order to participate in the communication represented by the connector.

At the implementation level, however, the realization of such a connector is complex and consists of many parts [51]. Consider again the RPC example, and suppose that this architectural RPC connector has been instantiated at the implementation level as a Java Remote Method Invocation (RMI) connection. Part of the connector implementation appears in the source code of the communicating components: the caller must initially obtain a reference to the remote callee, while the callee must initially register itself at a known location; at the call sites, the caller makes calls to a generated "stub" that stands in for the callee. Another part of the connector implementation appears in the stub itself (and perhaps a corresponding "skeleton" at the callee) and in communication libraries provided by the operating system and/or the middleware. Obtaining the reference to the callee may require the use of registration, naming, and lookup services; in this example, a process on the remote machine must be running the standard RMI registry provided by Sun, at a location known to the caller. In addition to these "running code" artifacts, RMI requires a security policy file (to permit the callee to accept incoming connections). Finally, independent of other concrete parts of the implementation, if some aspect of the architectural connector has been formally specified, this specification exists in some format (perhaps a text file) and should remain associated with the connector implementation.

Clearly, when one moves from architecture to implementation, it is no longer possible to point to a single entity and identify it as *the* connector. One might instead consider the connector implementation as a tuple $\{c, l, s, t, p, f\}$ (shown in Figure 3.1 and described below). The parts identified here may be spread across several files (or other concrete units) and mingled with other parts of the system, incidentally making generation or modification of connector mechanisms a difficult and nonlocalized task. Of the following kinds of connector parts, c, l, s are code artifacts and t, p, f are non-code artifacts:

c: Application-level code that appears within a component or compilation unit. This may be code at the point of communicating with another component (calling sites); also, there may be code necessary for the initialization and finalization of the connector, in "main()" or the equivalent part of this compilation unit.

l: Communication libraries, generated stubs, etc., below the application level.

s: Infrastructure services provided by, e.g., the operating system.

t: Data/tables such as the location of a communicating party.

p: A policy documenting the permissible use of these parts. For example, there may be a rule that library call x must be made before library call y. Another example is the security policy file in Java RMI; if a remote object (server component) in Java RMI chooses to incorporate the standard security manager, this file is consulted to determine whether to accept incoming connections.

f: Formal specification describing the connector's proper behavior.

Having considered communication-related connectors *in general*, in terms of architectural level abstraction and concrete implementation artifacts, let's look at more specific *kinds of* connectors. We'll return to the abstract level to do this.

When considered at high levels of abstraction, only a handful of basic forms of interaction, or *connector primitives*, exist. For example, the feature-based classification for architectural styles given by Shaw and Clements [50] identifies a small set of abstract connectors as a part of their style discrimination framework. Commonly used connector primitives include remote procedure call, implicit invocation and message passing (multicast and unicast asynchronous events), and dataflow.

A remote procedure call connector is asymmetrical, blocking, synchronous, and point-to-point. It is asymmetrical because there are two distinct kinds of roles, a *caller* and a *callee* (or definer). It is blocking, that is, the caller does not proceed until a response from the definer is received. It is synchronous because the caller can expect the definer to receive the call right away. It is point-to-point and the caller and definer are (to some extent) known to one another in advance. There are many examples of RPC implementations, including CORBA and Java RMI (Remote Method Invocation).

In contrast, an implicit invocation, or event-based, connector may be non-blocking, asynchronous, and broadcast (Clarke et al. [9] give a framework for the spectrum of event-based communication mechanisms). A sender sends an event and then continues operation without blocking; a listener receives an event at some point, not necessarily immediately, after it is sent; there may be zero, one, or many listeners and the sender generally does not know who is listening. Variants include publish-subscribe (such as Java Message Service), in which events are broadcast but any particular listener receives only the kinds of events to which it has "subscribed"; and message passing connectors (such as Unix IPC message queues; Java Message Service also offers this option), which are point to point; delivery guarantees also vary (and may commonly be weaker with publish-subscribe than with message passing connector implementations).

A dataflow connector is asymmetrical (has a *source* and a *sink*), asynchronous, and point to point. The source sends or writes data and the sink receives or reads it. Unix pipes are an example of a dataflow connector implementation.

Although only a few basic forms of interaction are enough to classify most forms of connector usage, connectors as they are actually implemented and employed are more complex variations on these forms; e.g., in constructing a software system, one might use not merely "an RPC connector" but "a Java-RMI-based RPC connector incorporating Kerberos authentication." The space of possible variations, and thus of "real" connectors, is huge. A central idea of this thesis is that many of these complex connectors can be viewed as a simpler connector plus one or more connector transformations; moreover, a set of transformations capable of producing a useful variety of complex connectors need not be unmanageably large, since a range of different effects can be achieved with transformations that are structurally similar. (This is similar to the Booch Components goal of separating policy and implementation; see section 2.4.3).

Consider the following two connectors. Connector A sends requests from a client component to a server component, performs on-the-fly compression, and caches results. Connector B sends requests from a client component to a server component; transmissions are digitally signed, and certain requests require authorization before they are accepted for processing. A and B address different concerns (bandwidth conservation versus security), but both of these dissimilar-seeming connectors can actually be constructed by using a *similar* composition of two connector transformations. Abstractly, one of the transformations that would be used applies a function f to data before it is sent and an inverse function f^{-1} when it is received (in one case, f compresses, while in the other, it appends a cryptographic checksum). The other transformation intercepts traffic and redirects it to a new component; the new component determines whether to forward the request to the server (in the case of a "cache miss" for A, or a successful authorization for B), or send a short-circuit response to the client (in the case of a "cache hit" for A, or an authorization *failure* for B). These and other transformations are described in greater depth below; but first, let's discuss what it means to be a connector transformation.

3.3 Connector Transformations

Returning to the architectural abstraction, in which a connector is a discrete design element, a *connector transformation* is a function from connectors (plus additional parameters) to connectors. Some transformations operate on a single connector type[1], while others operate on two or more connectors to produce some combination of them; in either case, the output is a single new connector type.

At the implementation level, where a connector is a tuple rather than a single entity, a connector transformation is a function from tuples (plus additional parameters) to tuples. Consider again the concrete artifacts listed in Figure 3.1. A connector transformation modifies one or more parts of an existing connector $C = \{c, l, s, t, p, f\}$, resulting in a new connector C'. For example, the transformation may add, remove, or modify lines of code in c (at the call sites or initialization/finalization). It may modify a library or stub in l or replace the library or stub with a different one that presents the same interface to the application-level code. The transformation may add a service in s or replace one service with another. It may add, modify, or remove entries in t. A transformation can affect the policy p, perhaps by adding new restrictions as a result of changing the assumptions needed to use the connector. A transformation also affects aspects of the connector that are described by the connector's formal specification f; this facet of connector transformations will be discussed in Chapter 5, and the impact of connector transformations on the rest of the tuple

[1]We will use the term *connector* loosely, when it is apparent whether a *connector type* or a *connector instance* is meant, to refer to either. "Java RMI" is an example of a connector type; "the Java RMI connector between this client and this server" is an example of a connector instance.

(c, l, s, t, p) will be revisited in Chapter 6.

3.3.1 Desiderata

As indicated in section 1.4.3, there are several desirable attributes for a set of transformations. The set must be manageable in size: *small*. It must be possible to produce a *wide variety* of *useful complex* connectors. To this end, the transformations must be broadly applicable (not specific to a particular connector type), non-domain-specific (not specific to the enhancement of a particular extrafunctional attribute), and, for the most part, composable with one another.

In order to constrain the set to an appropriate size while still being able to achieve adequate variety of the resulting connectors, the transformations must be of appropriate granularity: the person defining the transformations in the set must strike a balance between generality and power in determining the desirable degree of complexity of the individual operations. A set of very *simple, low level* operations may, owing to their greater generality, provide greater flexibility of the possible end results. However, additional effort will be required in composing larger numbers of these very small operations to create an augmentation that is actually useful. In addition, formal analysis of *individual* operations may be simpler, but (as greater numbers of operations are required to produce realistically desirable results) analysis of practical augmentations may actually be less tractable than in the case of a somewhat higher granularity of operations. Conversely, a set of very *powerful, complex* operations will require little or no composition effort. However, as these operations are highly specialized, they would require a significantly larger set of basic operations to provide the same level of breadth of possible results. To give a concrete example, consider a single powerful special-purpose operator, "add Kerberos authentication to this Java RMI connector"; it might be reusable for other forms of authentication but if someone wanted "retry with duplicate detection," a new single operator would be needed. In my approach these two different enhancements must be constructed from three smaller transformations[2], but, as witness of the greater flexibility of smaller transformations, they actually use the same three transformations (part of the difference being that they are used in a different order).

The next section presents a set of connector transformations that, I will argue,

[2]For more detail, Chapter 7 describes the use of connector transformations in adding these specific enhancements to actual Java RMI connectors.

meets these criteria.

3.3.2 A Set of Connector Transformations

The granularity of the set of connector transformations presented below is chosen to be sufficiently powerful so that even simple compositions of only one or two transformations begin to yield useful results; the set remains small enough to summarize on one page. I argue later in section 8.1.1 that this strategy for selecting transformation granularity provides a reasonable compromise between generality and power of individual transformations. Each connector transformation described here has points of variability (or parameters) that must be fixed in the process of applying the transformation. This enables the transformations to be generic in terms of extrafunctional enhancement and in terms of applicability to diverse connector types.

Finally, the means of deriving the set of transformations was chosen to enable the set to produce a useful variety of results. Recall that connector transformations are intended to address the lack of middle ground between, on one hand, "rolling your own" connector implementations, and, on the other hand, using only existing connector implementations; the former is difficult and expensive, and the latter may be insufficient to meet extrafunctional system requirements. A not-uncommon compromise that some have used in the past has been to modify (by hand) an existing connector implementation to add support for a specific set of neglected requirements; numerous diverse instances of such "complex connectors" exist. I extracted a small set of "generic" connector transformations by surveying and breaking down a wide variety of hand-modified complex connectors that appeared in publication, operating in reverse to arrive at a collection of transformations that are thus able to produce a wide variety of useful complex connectors. Chapter 4's informal construction of a number of well-known dependability-enhancing techniques, in terms of connector transformations, lends support to this claim (which will be further borne out by implementation generation results in Chapters 6 and 7).

Table 3.1 lists the transformations in this set and summarizes some of their key attributes: the *name* for this kind of generic transformation; the *effect*, in general terms, that this transformation has on a connector; the *parameters* that must be fixed upon when instantiating the generic transformation; and a few *typical uses* or domain-specific instantiations of the transformation.

Name	Operates on	Effect	Parameters	Typical uses
Data Translation	Connector	Apply function to payload	Function Location (sender/recipient)	Unit/format conversion Compress on the fly Encryption
Add Observer	Connector, new role	Eavesdrop w/o responding	Traffic copied to new observer component	Audit trail
Add Redirect	Connector, new role	Intercept, relay/reject	Traffic affected by the redirection	Authorize/confirm Add a cache
Add Switch	Connector, new role	Act in place of existing role	Traffic affected by the switch When to switch	Load balancing Backup server
Add Parallel	Connector, new role	Duplicates an existing role	Traffic copied to new component Merging responses	N-version Best-effort fidelity
Aggregate	Two connectors	A choice of connectors	When to choose How to choose	Protocol negotiation Bandwidth adaptation
Sessionize	Connector	Add state: record and replay	What to record When to record When to replay	Replay lost messages
Splice	Two connectors	A chimera combining dissimilar roles	N/A	Overcome protocol mismatch (between components)

Table 3.1: Summary of Connector Transformations

Data translation operates on a single connector. It applies a function (which must be specified) to the data being communicated by the connector. The "location" (the role of the connector) at which the function will be applied must also be specified.

The *Add role* transformations (the second through fifth rows in Table 3.1) operate on a single connector, plus a role. They introduce a new participant (the additional role) to the communication[3] represented by the connector. Some part (which must be specified) of the existing traffic will now be intercepted and sent to an additional destination or a different destination. The target of the redirection, and the responsiveness of this target, is constrained (differently) in each of these transformations: *Observer, Redirect, Switch,* and *Parallel replication.* In the *Add Observer* transformation, intercepted traffic is sent to the new role in addition to the original destination; the new role receives but does not respond to this traffic. In *Add Redirect*, intercepted

[3]Recall that this work focuses on communication-oriented connectors.

traffic is sent to the new role, which may choose to forward the traffic to the original destination. In *Add Switch*, intercepted traffic is sent to either the new role or to the original destination. In the *Add Parallel* transformation, intercepted traffic is sent to both the new role and the original destination.

Aggregate operates on more than one connector. It combines them into a choice of connectors. The points in the protocol at which it is permissible to switch to another connector, as well as the means of determining which connector to use, must be specified.

Sessionize operates on one connector. It adds state to a communication session by recording and later replaying some piece of data. The thing that is to be recorded must be specified, along with the appropriate moments to record, update, discard, or replay it.

Splice operates on more than one connector. It combines, from each of the original connectors, a subset of the connector's roles, to create a new, chimeric connector that *appears* to be partly one beast, partly another. For example, given binary (two-roled) connectors A and B with roles A_1, A_2 and B_1, B_2, a splice transformation might create a new binary connector with roles identical to A_1, B_2.

Each kind of transformation given in Table 3.1 is described in greater detail in the following sections. These informal descriptions cover the general intent and structural effect of the transformation; examples, including those given in the table; the parameters (points of variability); and the degree of transparency. I will use the word transparency to mean the maintaining of *protocol compatibility* between the new connector's interfaces and the kinds of component interfaces with which the original connector was compatible. (Practically speaking, in implementation a transformation that I here consider "transparent" may still have an effect on the communication of the system that is detectable to the components, e.g., because the transformation increases the resources used or affects the communication latency slightly; I will disregard this consideration when discussing transparency of a transformation.)

3.4 Data Translation

A *data translation* connector transformation (Figure 3.2) alters the data that is communicated, but leaves other aspects, such as the number of roles in the connector,

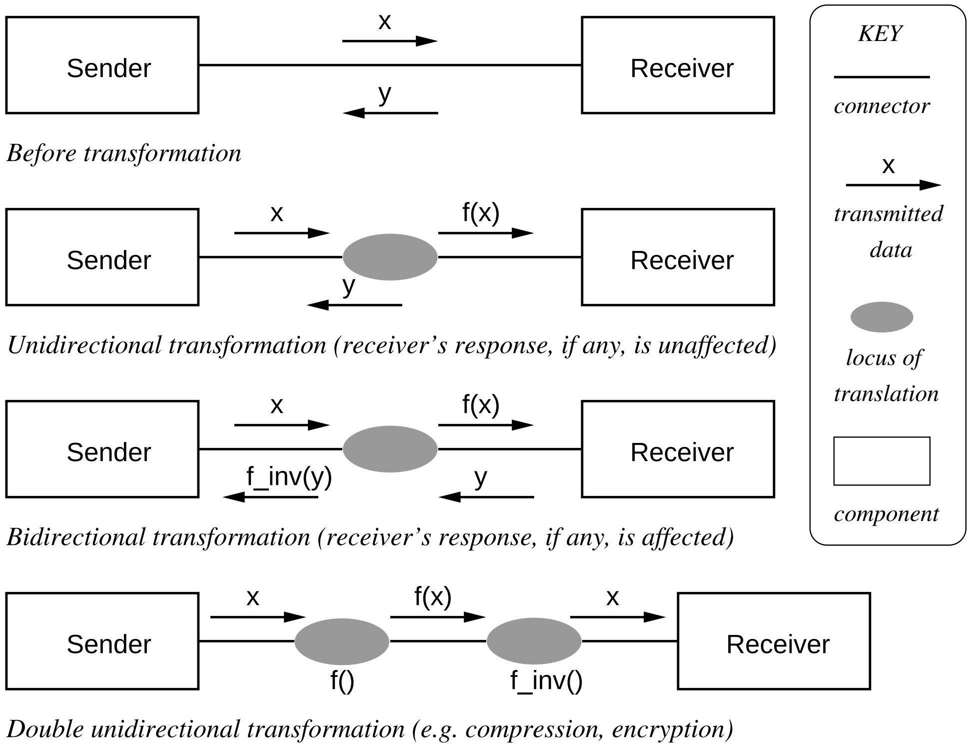

Figure 3.2: Sketch: Data translations in a two-role connector

unaltered.

In general, this kind of transformation intercepts data as it is transmitted or received and applies some function f to the data.

3.4.1 Examples of Data Translation

I give three examples here: mismatch resolution, on-the-fly compression, and encryption.

When two components agree on the semantics of a communication protocol but disagree on details such as the format or units of the data to be communicated, a data translation may be used to resolve the mismatch. To consider a trivial example, one component may provide data measurements in feet, while another component wishes to use measurements in meters.

A connector that compresses data on the fly can be created with matching data transformations at the sender and receiver roles. At the sender roles, a function f

compresses the data. At the receiver roles, its inverse function f^{-1} will be needed to uncompress the data. A more intelligent compression transformation may choose whether to compress the data or to send it uncompressed, based on size of the data and estimates of the time that would be required to compress, transmit, and uncompress versus the time required to transmit the uncompressed data (as in [27, 15]).

Similarly, one can construct an encrypting connector with a function f that encrypts data at the sender(s) and an inverse function f^{-1} at the receiver(s). Note that an additional transformation may be required (for example, to introduce a session key via "sessionize" or to obtain authentication via an added role) to complete the desired modification.

3.4.2 Parameters

Two aspects of the data translation may be varied.

The first is, as might be expected, the function f. In the examples above, selecting a different function naturally produces different results (perhaps differing in domain, as with compression for performance and encryption for security; or perhaps differing at a finer level of detail, such as different compression algorithms).

The second aspect that may be varied is the *location* at which the function is applied: at the sender of the communicated data, at the receiver, or (with a double transformation) both. Selection of location is affected by the format in which the data is desired to be for the actual communication. For example, in a broadcast connector, for a data translation used to resolve a mismatch of data format, one might choose a location at the sender if the receivers all use a single format, and otherwise a location at (some of) the receivers. Or, if there is a greater range of formats, one might convert to an intermediate format at the sender and convert again at the receiver.

Form

Informally, the form of this transformation is

Function $\times$ Location $\times$ Connector $\rightarrow$ Connector

Given a function (from data to data, not necessarily of the same type), a choice of location (a role of the connector), and a connector to operate on, the transformation

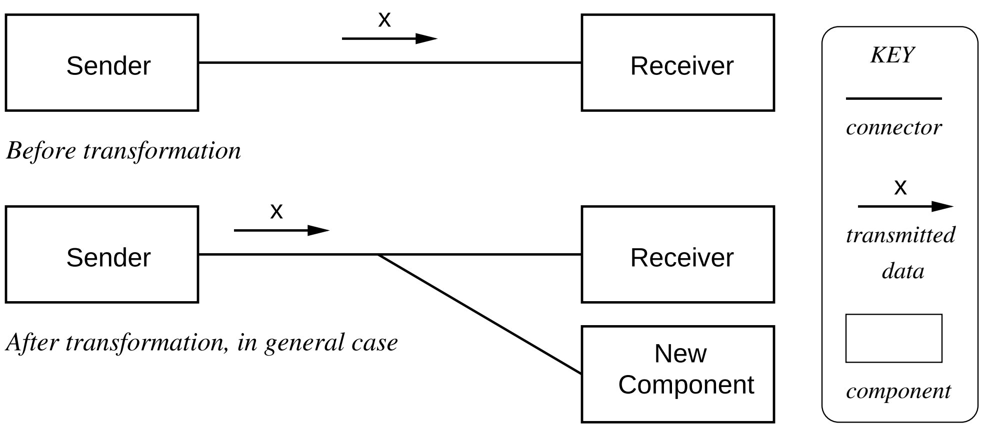

Figure 3.3: Sketch: Add-role in a two-role connector

results in a new connector.

Preconditions: The function must be injective on events communicated by Location.

For a double translation, such as the compression and decompression example, this transformation is applied *twice*: once (at the "sending" role) with the function f, and again (this time at the "receiving" role) with its inverse function f^{-1}.

3.4.3 Transparency

This transformation *may* be transparent to the communicating components (if, for example, a pair of transformations applying f and f^{-1} in succession are used). In the case of overcoming format mismatch, of course, complete transparency is not desired.

3.5 Commonalities of Add-Role Transformations

The Add Observer, Add Redirect, Add Switch, and Add Parallel connector transformations (Figure 3.3) all increase the number of roles in the connector. An added role may have a purpose and interface that is similar to a role that already exists in the connector; or the transformation may introduce a role that performs a new job, which may involve either pure observation or active participation in the communication.

In general terms, the parameters of these transformations are "what is intercepted" and they differ in "where it is sent." They can be considered partially instantiated forms of role addition that, by constraining a "where it is sent" parameter, cover an interesting range of *how* an added role may be permitted to interact with the rest of the connector.

Form

Informally, in the general case the form of the add-role transformations is

TrafficDirector $\times$ Role $\times$ TrafficSet $\times$ Connector $\rightarrow$ Connector

TrafficDirector's purpose is to intercept and redirect the traffic specified in TrafficSet. Its permissible redirection targets are constrained in each transformation; the interface supported by Role, the new participant, is constrained as well. Given a TrafficDirector, a role, a TrafficSet to intercept, and a connector, this transformation produces a new connector.

Preconditions: Events in TrafficSet must be members of the set of events communicated by the connector.

3.6 Add Observer

One might choose to add an *observer* role to a connector. This role does not actively participate in the communication: it eavesdrops on traffic between other participants.

3.6.1 Examples of Observer

I give four simple examples here: an auditor, a performance monitor, a workload recorder, and a displayer.

First, an observer may be added to log traffic for security reasons, providing an audit trail. Second, with timestamps an observer may be used as a crude application-level performance monitor. Third, an observer can be used to record a characteristic workload for the system, which might be replayed later for simulation or debugging purposes. Finally, the observer component may make its observations visible in real time to a human, giving a window into the communication, which may be useful for debugging, understanding of dynamic behavior, demonstrations, etc.

3.6.2 Parameters

In the "add observer" transformation, the parameter of what is intercepted remains unbound. The parameter of where intercepted communications are sent is fixed: intercepted communications are *copied* and the copy is sent to *the new role* (the observer component).

Form

Informally, the form of this transformation is

$\mathit{TrafficDirector} \times \text{Role} \times \text{TrafficSet} \times \text{Connector} \rightarrow \text{Connector}$

Preconditions: TrafficDirector should send a copy of traffic in TrafficSet to Role. Role, similarly, should accept incoming communication in TrafficSet (but if it behaves strictly as an observer, Role does not have outgoing communication on this connector and TrafficDirector should not need to forward any traffic generated by Role).

3.6.3 Transparency

This transformation is transparent to the previously existing roles of the connector.

3.7 Add Redirect

Another manner of participation in existing communication is for an added role to essentially intercept some of the traffic generated by existing role(s), and to relay, bounce, or replace these transmissions. That is, a subset of traffic is *redirected* to this new role for its appraisal, before (perhaps) being sent to its original destination.

3.7.1 Examples of Redirect

A redirecting transformation may be used to introduce a layer of authorization, confirmation, or other forms of sanity-checking, described in the two following examples.

Perhaps certain services offered by a component should only be accessible to authorized clients, e.g., for security reasons or even to enforce a design constraint. Client requests for these services can be redirected to a component that checks authorization and either relays the request to the server or reports an error to the unauthorized client.

Again suppose that some actions or situations require human intervention to judge their appropriateness or safety. For example, a redirect transformation may intercept questionable or irrevocable actions and pop up a familiar "are you sure?" dialog which requires human input to either continue or abort the action.

A redirection can also be used for other purposes, such as to improve performance by introducing a simple cache to which some kinds of traffic are redirected.

3.7.2 Parameters

In the "add redirect" transformation, the parameter of what is intercepted remains unbound. The parameter of where intercepted communications are sent is fixed: intercepted communications are *redirected* to *the new role*; the component then decides whether to relay the communication to the original intended recipient, or whether to send a "short circuit" response back to the sender.

Form

Informally, the form of this transformation is

$\mathit{TrafficDirector} \times \text{Role} \times \text{TrafficSet} \times \text{Connector} \rightarrow \text{Connector}$

Preconditions: TrafficDirector should redirect the traffic in TrafficSet to Role, and it should allow Role to relay or to respond. Role, similarly, should accept incoming communication and, if a response is expected, either relay or respond.

3.7.3 Transparency

This transformation will be transparent to the sender in certain situations: specifically, in situations where the added participant responds only in ways that are already familiar to the sender (e.g., a cached result, or an exception type that the sender already handles). This transformation is transparent to the receiver (who will simply see less traffic).

3.8 Add Switch

When an added role has a purpose and interface similar to an existing role, it may be intended to operate either as a replacement for the existing role (so that both are not in operation at the same time, except perhaps as a passive observer) or as a replicant of the existing role (so that both are in operation at the same time). The transformation required for the first case introduces a *switch*; I will refer to the transformation required for the second case as *parallel* replication (discussed below in 3.9).

An *add switch* transformation adds a role R' that is a companion to an existing role R. Either R or R' is active at any point in time. When R is not active, its traffic is redirected to R' which may respond in place of R.

3.8.1 Examples of Switch

I give three examples here: adding a stateless backup server, load balancing, and variable fidelity computation.

One example of a switch is the introduction of a "backup" component C' which will replace the "primary" component C whenever failure (e.g., nonresponsiveness, erroneous results) of the primary is detected. In this case, the trigger is the detection of failure. Subsequent recovery of C should, when announced or detected, trigger a switch back from C' to C.

One might also use a switch transformation to introduce load balancing at a high level; if C and C' are stateless, traffic may be directed to each in alternation without affecting the result.

Another example might be the replacement of a component with high resource requirements with a low-fidelity component that has lower resource requirements. This shift may be triggered by a change in the available resources in the environment (processing power, bandwidth, etc.) or a change in the quality of service requirements (for example, acceptable latency may be lowered, or desirability of accurate results may be increased).

3.8.2 Parameters

In the "add switch" transformation, the parameter of what is intercepted remains unbound. The parameter of where intercepted communications are sent is fixed: intercepted communications are sent to *either* the new role or the existing role for which it is a replacement, depending on which is "active."

Form

Informally, the form of this transformation is

$\textit{TrafficDirector} \times \text{Role} \times \text{TrafficSet} \times \text{Connector} \rightarrow \text{Connector}$

Preconditions: TrafficDirector should send the traffic in TrafficSet either to Role or to its original destination. (The actual decision-making process is not specified here, that is, the event or criterion that causes TrafficDirector to change from one destination to the other.)

3.8.3 Transparency

This transformation is likely to be transparent to the sender (since the new and old roles have similar behavior) and is transparent to the recipient (though, as with the redirect form, it may see less traffic).

3.9 Add Parallel

An *add parallel replication* transformation adds a role R' that is a companion to an existing role R. Both R and R' are active at the same time. Traffic sent to R is also sent to R'. If a single reply is expected, their replies must be intercepted and reconciled by the connector into a single reply.

3.9.1 Examples of Parallel Replication

I give two examples here: n-modular redundancy and "best-effort" variable fidelity computation.

Multiple instances of a single component may be introduced as an effort to increase dependability. For example, the output of three instances (running on separate hardware) may be reconciled by voting, so that errors experienced by a single instance can be detected and masked.

Multiple versions of a single component that differ in the "fidelity" and latency of their output may be used for performance reasons. Here the connector may reconcile replies by selecting the best-fidelity response that arrives before the deadline (or, within a time interval T).

3.9.2 Parameters

In the "add parallel" transformation, the parameter of what is intercepted remains unbound. The parameter of where intercepted communications are sent is fixed: intercepted communications are copied and sent to both the new role and the existing role(s) that the new role replicates. Multiple responses (if any) must be intercepted and reconciled.

Form

Informally, the form of this transformation is

$\mathit{TrafficDirector} \times \text{Role} \times \text{TrafficSet} \times \text{Connector} \rightarrow \text{Connector}$

Preconditions: TrafficDirector should send the traffic in TrafficSet to both Role and its original destination.

I do not here constrain the process that deals with the responses; both Role and the original destination may respond, and it may be desirable to reconcile the responses so that the sender always receives at most one response.

3.9.3 Transparency

This transformation is transparent to the original participants (if the responses are reconciled).

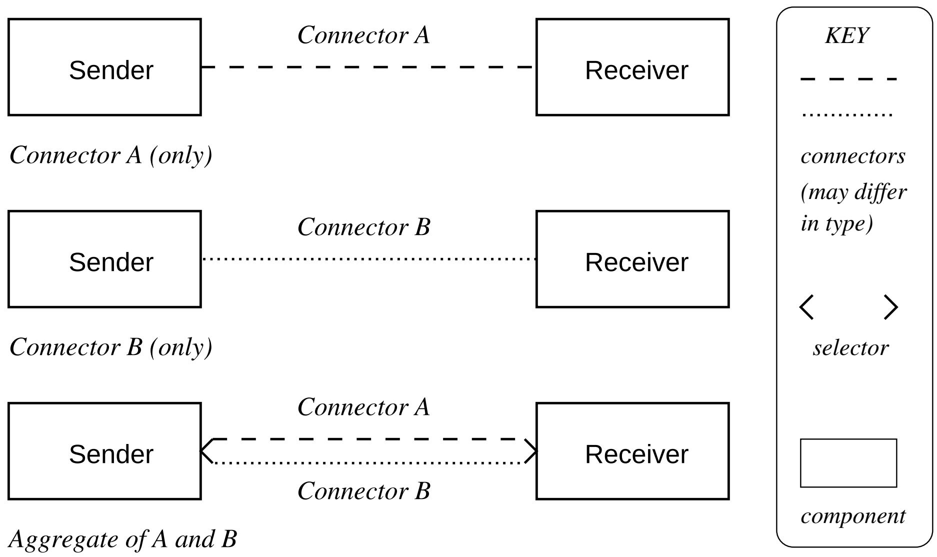

Figure 3.4: Sketch: Aggregate of two two-role connectors

3.10 Aggregate

In an *aggregate* connector transformation (Figure 3.4), two (or more) connectors are combined into a single composite connector that may behave like either of the original connectors. At any particular time, at most one of the aggregated connectors is "active." The decision of which aggregated connector to use is made initially; in some situations it may also be permissible to revisit the decision at certain constrained points in the communication, which may result in deactivating the currently used aggregated connector and activating a different one, if the conditions on which the initial decision was based have altered.

3.10.1 Examples of Aggregate

I give two examples here: initial negotiation and dynamic adaptation.

Suppose that two components wish to begin communicating with one another but have not agreed on a specific protocol in advance; rather each supports some subset of an agreed-upon set of protocols. An aggregate connector, composed of connectors that implement members of this set of protocols, can be used to negotiate initially a mutually agreeable protocol (if one such exists). In this example, the decision of which aggregated connector to use is made only once, when the communication is initialized.

Another example, in which the active connector may be changed on the fly to another connector, is one in which a set of similar connectors that differ in extrafunctional attributes (e.g., one has better performance, another has better reliability) are available for use. Based on the prevailing conditions, the "best" connector is selected; when conditions change, e.g., an increase in error bursts in the transmission channel requires an increase in fault tolerance at the expense of performance, the current connector is no longer "best" and communication is switched over to a more appropriate connector.

3.10.2 Parameters

There are two points of variability for the aggregate transformation.

The first is *when* it is permissible to choose a different internal connector. At one end of the spectrum of dynamism, the connector may be selected or negotiated at startup and then never changed. Or, there may be specific intervals in the communication protocol of each internal connector at which it is "safe" to stop using that connector and begin using one of the alternative connectors.

Note that the locations of safe points within a connector are dependent on the connector type and on its communication protocol. The work described in this thesis does not provide facilities for determining what the safe points are. The implementation generation tool described in Chapter 6 offers a restricted set of permissible locations in the protocol (these include: at initialization; prior to sending a request; and, in connectors where a response can be expected, upon receipt of a response), of which it is the tool user's responsibility to choose a subset. Similarly, the formal template for the aggregate transformation, provided in section 5.4.3, assumes that the user of the formalism has specified the points at which it is permissible to transition from the process describing one connector's protocol to the process describing another connector's protocol.

The second point of variability is *how* the "best" connector is to be determined. This selection function is domain specific (like the function parameter in the data translation transformation); that is, the definition of "best" necessarily depends on the extrafunctional attributes that this transformation is being employed to enhance.

Form

Informally, the form of this transformation is

Selector $\times$ SelectionPoints $\times$ Connector $\ldots \times$ Connector $\rightarrow$ Connector

Given a Selector (a decision-making process to pick the current "best" connector from the available set), a choice of Dynamism, and two or more connectors, this transformation produces a new connector.

Preconditions: SelectionPoints must be a subset of the "safe" points, which are determined by the connectors being operated on. There must be a correspondence between roles of the input connectors.

3.10.3 Transparency

If the connectors that can be selected are transparent to the participants (that is, a participant can use connector A or B without being modified), and the points of changeover, which are asserted to be safe, are actually safe, then one can expect the aggregation to be transparent.

3.11 Sessionize

A *sessionize* transformation (Figure 3.5) introduces state into a connector that (with respect to that state) was previously stateless. The value of the state is updated at the start of a "session" and this value may be used (and perhaps updated or incremented) within that session.

3.11.1 Examples of Sessionize

Consider a connector that uses an imperfect channel in which messages don't always arrive intact. There are several well-known techniques (such as timeouts and error detection codes) that can be used to detect missing or corrupted messages; once the problem is detected, one might like to try re-sending the affected message. A sessionize transformation can be used to *record* a message as it is sent out and subsequently

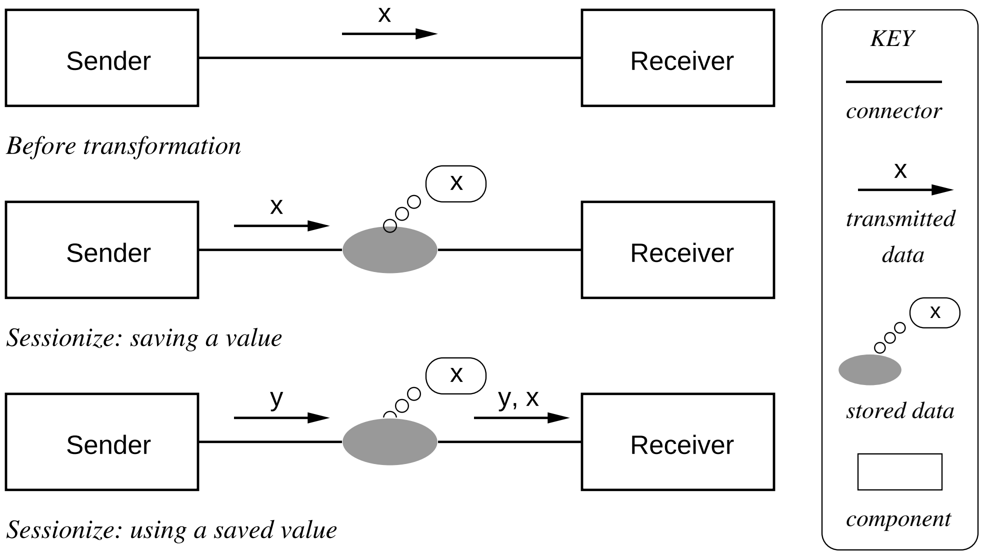

Figure 3.5: Sketch: Sessionize, saving and replaying

replay it if a problem is detected (or erase it if no problem is detected; for example when an acknowledgement is received).

In addition to combining this sessionize transformation with the transformations that enabled detection of the problem, one might also plan to add a data translation that stamps each message with a unique identifier (before it is recorded by the sessionize transformation) and (if the acknowledgement, rather than the message, is lost, which cannot be distinguished by the sender) filters out duplicate messages at the receiver.

3.11.2 Parameters

The parameters for this transformation are: what to record, when to record (or update or discard) it, and when to replay it.

Form

Informally, the form of this transformation is

Recorder $\times$ Replayer $\times$ Connector $\rightarrow$ Connector

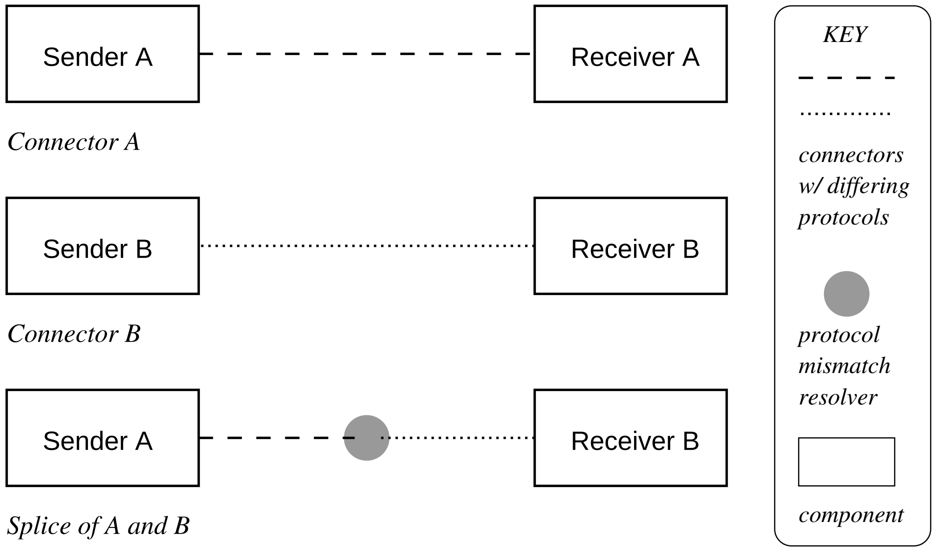

Figure 3.6: Sketch: Splice of two two-role connectors

Given a Recorder (which determines what to record and when), a Replayer (which determines when to replay available recorded state), and a connector, this transformation produces a new connector.

Preconditions: Any event recorded by the recorder must be a member of the set of events communicated by the connector.

3.11.3 Transparency

This transformation may require composition with another transformation (as in the "replay" example above) to achieve transparency.

3.12 Splice

In a *splice* connector transformation (Figure 3.6), two connectors that implement somewhat different protocols (and, as a result, are likely to have differing interfaces) are combined into a single composite connector that presents interfaces from both of the original connectors; e.g., if two binary connectors are spliced, the result will be a binary connector with one role from each of the original connectors. Splicing two arbitrary connectors may be difficult or impossible, such as when one connector's protocol requires access to more information than is communicated by the other connector's protocol.

3.12.1 Examples of Splice

Splicing is performed to repair mismatch. Yellin and Strom [64] show that some connector protocols may be repaired through the use of "component adaptors", which incorporate a state machine; they give a means of describing which messages can be sent and received by a particular component, and a way to check compatibility. I give a simpler example here, which will be revisited in Chapter 5.

Consider two connectors C_1 and C_2. Each has the same purpose: to transmit the same set of sensor readings from some component S (which reads the sensors) to some receiving component R. Their protocols are similar, except that in C_1 the set of readings is bundled into a single transmission; in C_2, each reading is sent separately. Suppose that S expects to use C_1 and R expects to use C_2 (or vice versa); a spliced connector would resolve the mismatch and enable S and R to communicate. In order to splice these two connectors, the transformation must either expand the bundled transmission into separate transmissions or compact the separate transmissions into a bundled transmission.

3.12.2 Parameters

This transformation depends on the connectors supplied to it and the degree of mismatch between them. At minimum, one must identify which of the original connectors' roles should appear in the new connector and provide a policy that clarifies any underconstrained aspect of the mismatch resolution (in the example above, suppose that S sends unbundled sensor readings and some readings are sent more frequently than others; it would be necessary to clarify how often the spliced connector should send R a bundled reading).

Form

Informally, the parameters of this transformation include

Connector $\times$ Connector $\times$ RoleSet $\times$ Policy $\rightarrow$ Connector

RoleSet indicates which roles of the original connectors are to be carried over into the final connector.

3.12.3 Transparency

"Splice", like the mismatch-resolving example of the data translation transformation, is inherently not transparent; it cannot seamlessly replace a connector from which it was created. However, each of its roles should present an interface that is compatible with one of the original connectors' roles.

3.13 Summary

In this chapter, I defined a small set of basic structural transformations that can be applied compositionally to simple connectors. The "data translation" and the "sessionize" transformations operate on a single connector. The "add observer", "add redirect", "add switch", and "add parallel" transformations operate on a single connector plus a role which is to be added to the connector. The "aggregate" and "splice" transformations operate on multiple connectors, combining them into a single connector. Each transformation has been described informally here and accompanied with simple scenarios of its possible use in a variety of domains.

The transformations outlined here can be composed to produce more complex modifications. A transformation may be composed with itself (e.g., two data transformations f and f^{-1}) or with a different kind of transformation. I will next illustrate (within the specific domain of dependability enhancements) how compositions of these transformations can produce a "wide variety of useful complex connectors" as claimed. The ordering of the composition will in some cases affect the outcome; formal determination of commutativity of transformations is discussed later in section 5.3.4. Section 6.4 will discuss issues that must be considered at the implementation level when composing transformations.

Chapter 4

Coverage Within A Domain

4.1 Overview

I have described a small set of connector transformations; but is this small set sufficient to construct, within a particular domain, the commonly desired enhancements to a connector? To answer this question, this chapter introduces an investigation of the coverage of these transformations. I show the degree of coverage of a set of domain-specific enhancements in the domain of dependability, first by describing a commonly understood set of enhancements (without reference to connector transformations) and then by indicating how to select connector transformations to construct these enhancements. I focus here on illustrating a degree of coverage *within* a domain; section 8.3 will evaluate the particular choice of domain and discuss the issue of coverage *across* domains.

Readers who are already familiar with dependability may wish to skip sections 4.3 and 4.4, which provide a more detailed background in dependability (without reference to connector transformations) in preparation for section 4.5.

4.2 Dependability: A Categorization Framework

The motivating example seen in section 3.1.1 enhances the dependability of a connector. A single such example would show that it is *possible* for connector transformations to achieve a domain-specific enhancement in the domain of dependability. To show that a *useful range* of such dependability enhancements are possible, we first must look at a categorization of dependability enhancements so that later we can consider the degree of coverage provided by connector transformations.

Avizienis, Laprie, and Randell [7] define four standard categories of techniques that people can use to achieve greater dependability in software: fault prevention, fault tolerance, fault removal, and fault forecasting. Fault prevention techniques can be used during design and implementation of software to produce software with fewer faults. Fault tolerance, which will be discussed further below, is intended to keep software running in spite of faults; the techniques are used when the software is designed and implemented and may come into play when the software is in operation. Fault removal techniques include verification and validation, which can be used during design and implementation of software to discover faults, and maintenance techniques, which can be used to remove faults from (i.e., debug) software during its operational life. Fault forecasting techniques are used to guess how many faults are in a software system and, probabilistically, what their effects will be (that is, how reliable is the system likely to be).

Fault tolerance techniques are further decomposed by Avizienis et al. as follows (see Table 4.1 for a summary). When the software is in operation, an error (the symptom of a fault) is discovered using **error detection** techniques. The detection of an error may then trigger various **recovery** techniques that try to return the system to a useful state: first, **error handling** recovery techniques attempt to get rid of the *error* so that the software can continue running properly, and then a sequence of **fault handling** recovery techniques may attempt to get rid of the actual *fault* itself (the design flaw, or bug, that the error is a symptom of) by localizing it in the software implementation and preventing that piece of code from running again. The essence of error handling is to get from a bad (erroneous) system state to a good (error-free) state. Error handling techniques can be subcategorized into three kinds: those that *roll back* to an old good state that was recorded before the error, those that *compensate* for an error by simply having enough redundancy in the state itself to eliminate some classes of errors, and those that *roll forward* to a new good state. Fault handling involves a sequence of smaller techniques: the error must be *diagnosed* to determine the kind of fault that caused it and to locate the component that contains the fault; once located, the faulty component must be *isolated*, that is, removed from service; the software system may then be *reconfigured* and *reinitialized* to compensate for the loss of this faulty component.

Phase	Options	Techniques
Error Detection	Acceptance Tests (AT) Comparing Variants	different accuracy, inverse function, certification trail, boundary conditions, rate of change, timing, uncaught exceptions
Error Handling	Roll Backward Roll Forward Compensation	checkpointing community error recovery mask with majority vote, forward error correction (FEC)
Fault Handling	(diagnosis, isolation, reconfiguration, reinitialization)	remove a variant

Table 4.1: Fault Tolerance Building Blocks

Having considered this framework for categorizing the building blocks (error detection, error handling, and fault handling techniques) of fault tolerance techniques, let's look at a collection of more specific well-known fault tolerance techniques described in [35]. Table 4.2 (which appears later in section 4.4) summarizes the relation between these approaches and the building blocks of Table 4.1, and indicates whether variants are employed sequentially or in parallel. For distributed systems, three broad approaches are described, each with a number of variations: recovery blocks (RB), N-version programming (NVP), and N self-checking programming (NSCP). Laprie et al. [33] observe that these correspond to three well-known hardware-level techniques: RB is similar to standby sparing or "passive dynamic redundancy", NVP is inspired by n-modular redundancy or "static redundancy", and NSCP is inspired by "active dynamic redundancy".

These three approaches (RB, NVP, and NSCP) add redundancy to components of a software system by having more than one *variant* (more than one implementation) of some particular piece of the system in order to tolerate faults within implementations of that piece. Each approach incorporates an error detection technique and an error handling technique. Their error detection techniques can be divided into two kinds:

those that use *acceptance tests* and those that *compare results of variants* [33]. Their error handling techniques may be rollback, rollforward, or compensation-based.

I will first go over these error-detection and error-handling building blocks before describing the larger techniques and their variations or extensions. I will then describe how connector transformations may be used to assist in the incorporation of these techniques to software systems.

4.3 Building Blocks

Let's take a closer look at the techniques in Table 4.1.

4.3.1 Error Detection Techniques

Laprie et al. [33] discuss two main approaches to error detection: one is the use of acceptance tests, and the other is the comparing of results of two variants.

Acceptance Tests

Acceptance tests are application specific; that is, an acceptance test incorporates knowledge that in some degree depends on the function being computed, in order to detect some kinds of incorrect results. It is important for an acceptance test to be faster and *simpler* (so that it is less likely to be the source of a detected error) than the original computation. The inputs to an acceptance test may include timing information, past results, the computation's inputs, and intermediate results, in addition to the current result that is to be tested. Kinds of acceptance tests include the following.

Different accuracy — Given a value that is the result of a (lengthy, precise) computation, there may be a much faster to run and simpler to implement, but less accurate, computation that will indicate whether the given value is in the right ballpark. If the more precise result differs greatly from this "back of the envelope" calculation, the expectation is that the more precise result is probably incorrect.

Inverse function — For some applications, the function being computed has an inverse that is easy to compute. For example, the original computation may be to factor a number, and the result can be checked by a trivial multiplication. If the outputs of the inverse function do not match the inputs to the original function, the expectation is that the given result of the original function is incorrect.

Certification trail — In this technique, the original computation must be designed to produce a "trail" of intermediate results that can be used to confirm the final result more quickly and easily than a recomputation. The acceptance test's inputs in this case are the final result plus the trail.

Boundary conditions — The results of some functions are naturally bounded so that a result falling outside a boundary is automatically suspicious. For example, a computation whose output is supposed to represent an index in an array should not produce a negative value nor a value that is definitely greater than the size of the array.

Rate of change / physical laws — Some results may be rejected due to their apparent violation of physical laws. This is similar to the testing of boundary conditions. For example, given timing information and a previous result, a new result that differs too greatly (indicating, e.g., an impossible acceleration) would be detected as suspicious.

Timing — A "watchdog" timer may be used when there are approximate bounds on the expected time to compute a result. For example, if a particular computation ought to take less than t time and has taken longer without yet producing a result, one might suspect that something has silently failed.

Exceptions — Trivially, an uncaught exception thrown by the component whose results are to be tested is a suspicious result.

Comparing Variants

As a means of error detection, comparing variants is fairly self-explanatory. It is necessary to have at least two variants and a *decision algorithm* to use to compare them. If the results of the variants differ, one would expect that something is wrong, though there are some application-specific subtleties. Some computations may be imprecise in nature, so that results within epsilon of each other may be considered "close enough". Some computations may have more than one right answer according to the specification; for example, a lookup function that returns the index of a given value in an array, in the case when the value appears more than once, might be permitted to return the index of any of its appearances. In cases such as these, a simplistic "$a = b$" comparison would not be sufficient; a slightly more complex comparison such as "$||a - b|| \leq \epsilon$" would work for the case of imprecise computations, but not for the case of multiple correct answers.

4.3.2 Error Handling Techniques

After an error is detected it is desirable to *recover* the system to a consistent state. Many approaches make use of backward recovery. Other possibilities are forward recovery and compensation.

Backward Recovery

In backward recovery, the state of a variant is recorded in a checkpoint and can later be "rolled back" by resetting the state to this recording of a previous state.

Many of the approaches that I will describe below require the ability to checkpoint (to some extent) the state of a component at specific points before executing a questionable piece of code whose results will subsequently be submitted to some means of error detection. If an error is detected, these approaches need to be able to roll back the execution of the component to this checkpoint, so that a different piece of code (hopefully producing a better result) can be executed instead. (Provision for this kind of backward recovery must be included by the programmer of the component.)

Rollback in a distributed setting can snowball (known as the "domino effect") if components have communicated during the block of code that is to be rolled back on one component; there are rollback-coordination techniques that attempt to mitigate this problem.

If *backwards recovery* via rollback is not possible, some of the approaches discussed in [35] can still be used either with *forward recovery* or with *degradation* (compensation plus fault removal).

Forward Recovery

Whereas backward recovery "rolls back" to a previous state that was known to be consistent, forward recovery attempts to move forward from a bad state to a *new* good state.

For example, in n-version programming, several versions of a component may have executed on different hosts, and one may have produced an incorrect result; with forward recovery, the erring version might be informed that its result is incorrect and given the consensus result so that it can correct its internal state, get back on track, and hopefully continue execution. (Provision for this kind of forward recovery must be included by the programmer of the component.) This example is known as "community error recovery" [8].

Compensation

Compensation makes use of redundancy present in the current state to mask an error in the state. In the case of n-version programming, for example, if a majority of results of variants agree, they could be used to mask the differing results of a minority of presumed-faulty variants. Some techniques that deal with tolerating faults in communication channels are compensation techniques: for example, forward error correction codes add redundancy to a message before it is transmitted, and the receiver uses this redundancy to tolerate 1-bit (or more, depending on the code) errors that were introduced during transmission.

Degradation

If it is not possible to "roll back" for a backward recovery, and if it is not possible to perform a forward recovery for the erring component, Laprie et al. [33] note that (for approaches based on NVP or NSCP) the erring variant could be designated as "failed" so that execution can continue in a degraded mode.

For example, in n-version programming, several versions of a component may have executed on different hosts, and one may have produced an incorrect result; if there is no provision for rollback or for community error recovery, execution could still continue with the versions that are correct. If we drop from three versions to two versions in this example, we can still detect a single error if the two versions differ in their results, though we can no longer detect which result is likely to be correct.

Returning to the Avizienis et al. classification, this simple form of degradation is actually a *fault handling* technique rather than an *error handling* technique. The error handling in the NVP example is essentially *compensation*, as the redundancy of the variants enables us to ignore the part of the state that came from the faulty variant, leaving only the "good" part of the state that came from the two non-faulty variants. The fault handling includes a diagnosis technique (detecting which variant is faulty, in this example by comparing results of three variants), followed by isolation, reconfiguration, and reinitialization, which in this example were lumped together as "dropping from three versions to two versions."

With *graceful degradation*, a system as a whole can "shed functionality" and continue to offer some degree of utility, even when some non-replicated parts of it have failed. Shelton et al. describe a scalable architecture-based approach [52] in which components are grouped into subsystems according to their input/output dependencies; this means of partitioning is then used in enumerating valid system configurations (situations in which enough components are functioning to provide some utility).

4.4 Approaches to Fault Tolerance

Lyu [35] identifies three kinds of distributed approaches to fault tolerance that are well-known in software: those that are based on recovery blocks (RB), N-version programming (NVP), and N self-checking programming (NSCP). In addition a number of extensions and hybridizations of these three approaches are identified. I list them here, taking note of which of the previously discussed error detection and error handling techniques (as categorized in section 4.3) are used. Table 4.2 gives a summary, and indicates (in the "Timing" column) whether variants are employed sequentially or in parallel.

Recovery blocks

The original recovery block approach was applicable to sequential systems (from the perspective of a distributed system it might occur *within* a component) but has been extended to distributed systems. In the original approach, a sequential block of code designated a "recovery block" has one or more ordered variants called *alternates*, plus an *acceptance test*. When execution enters the recovery block, a checkpoint must be made. The first alternate is executed, then an acceptance test is performed on the results of this execution; if a failure is detected, the block is rolled back to the checkpoint and the next alternate is executed and tested, etc. There is no guarantee that an alternate will be executed, so the programmer must take care that alternates

Approach	Error Detection	Error Handling	Fault Handling	Timing
Recovery blocks	Acceptance Test (AT)	backward	—	sequential
Primary/shadow pair	AT	compensate	shadow becomes primary	parallel
Standby sparing	AT	compensate	spare takes over	sequential
Distributed RB	AT	backward	shadow becomes primary	both
Consensus RB	Compare; AT if no consensus	—	—	parallel
Retry blocks with data diversity	AT	backward	—	sequential
Retry	AT [timer] or FEC	forward	—	sequential
Self-configuring optimal programming	Compare and/or AT	—	select other variants	both
N-version programming	Compare	fwd. or compensate	eliminate variant	parallel
N-modular redundancy	Compare	compensate	eliminate host	parallel
N self-checking programming	Compare and/or AT	backward	shadow becomes primary	parallel

Table 4.2: Fault Tolerance Approaches

do not retain any state.

Randell and Xu [46] identify four kinds of failure detection in a recovery block: failure of the acceptance test, failure to terminate (detected by a watchdog timer, which I describe above in acceptance tests), "implicit" error detection such as division by zero, and an exception thrown by a nested recovery block (which would occur if the nested block's variants all failed the nested block's acceptance test).

In the RB approach, error detection is via acceptance test and recovery is via backward recovery. If backward recovery is infeasible (e.g., in a real-time system), [46] observes that forward recovery may be possible (if other components of the system have been implemented to support it) by sending a “disregard previous output” message from the component to the usual recipients of its output, whenever the component’s acceptance test fails.

Extensions of the recovery block (RB) approach that are identified in [46] are: distributed recovery blocks, consensus recovery blocks, retry blocks with data diversity, and self-configuring optimal programming. I will touch on each of these in turn, first discussing two hardware fault tolerance techniques that are simpler relatives of recovery blocks and distributed recovery blocks.

Primary/Shadow Pair

“Primary/shadow pair” (PSP) is a technique for hardware fault tolerance, of which the distributed recovery block technique is an extension [32]. In PSP there is a primary host and a hot standby “shadow” host. Both run the same software, in parallel, with the same input data; each checks its own output with an acceptance test, and the shadow listens to the primary’s output to detect possible silent failures.

If the primary fails, the shadow takes over; meanwhile, if the primary is still alive and not crashed, the primary retries the same computation and (if this recovery is successful) becomes the shadow.

Standby Sparing

This is a technique for hardware fault tolerance that corresponds to recovery blocks [33]. Here there is a primary host and a passive standby that becomes active if the primary fails; where PSP has the primary and shadow running in parallel, standby sparing has them running sequentially.

Distributed Recovery Blocks

The distributed recovery blocks (DRB) technique is an extension to recovery blocks that is based on the PSP technique for hardware fault tolerance [32]. In contrast to PSP, where the primary and the shadow both run the same software, in DRB the software has a *primary version* and an *alternate version*. The primary host runs the primary version, and the shadow host runs the alternate version. There is also an acceptance test (available on each host).

If the primary version fails the acceptance test on the primary host, the primary host will recover by rolling back and running the alternate version. Meanwhile, the shadow host takes over as the new primary host (and subsequently runs the primary version), provided of course that its result passed the acceptance test. The old primary host becomes the new shadow host and continues to run the alternate version of the software.

Somewhat symmetrically, if the alternate version fails the acceptance test on the shadow host, the shadow host will recover by rolling back and running the primary version of the software (the primary host remains primary and the shadow host remains shadow).

Consensus Recovery Blocks

This extension of RB is a hybrid that includes ideas from the n-version programming approach in an effort to mitigate the weaknesses of each (the importance of the acceptance test in RB and the case of multiple correct answers in NVP). In this approach, there is a set of ordered alternates, an acceptance test, and a comparator. The alternates are executed and pairs of results are *compared* first. If there are two matching results, they are assumed to be correct. If there is no matching pair, the results of the alternates are submitted (in order) to an acceptance test, until one passes and is assumed to be correct.

Retry Blocks With Data Diversity

In this approach there is one variant, an acceptance test, and one or more *data re-expression algorithms*. If the acceptance test fails, the variant is rolled back, its inputs are re-expressed (using the re-expression algorithm), and the variant is executed again, producing a different result which might pass the acceptance test this time.

Retry (a degenerate case)

Consider a degenerate form of recovery blocks in which the variants are identical; this is equivalent to a degenerate form of "retry blocks with data diversity" without the data re-expression algorithms. Though not discussed in [46], which focuses on components rather than communication, this technique can be useful in cases where a fault is *transient* as with an imperfect communication channel; in this case, a retry is essentially a retransmission.

Error detection in this approach may be via a watchdog timer (to detect dropped transmissions), an acceptance test if applicable, or an error detection code (to detect errors introduced in transmission). Error handling is generally by rolling forward. If the destination component did receive the original transmission (and its response was lost instead) then prior to error handling the sender and destination are in an inconsistent state with respect to one another; the destination has moved forward and the sender needs to catch up. It is possible to roll the sender forward to a state consistent with the destination, if the destination *detects duplicate transmissions* and *resends* (rather than recomputes) the corresponding response. The sender may tag each transmission with an identifier to simplify identification of retransmissions.

A degenerate case of this "retry" technique is to always send every transmission n times (e.g., 2 times, 3 times).

Self-Configuring Optimal Programming

In this hybrid approach, there is a pool of possible variants, the execution *selects* some variants, and the results are compared and/or submitted to acceptance tests. If it fails, some of the remaining variants are selected and executed. This is a trade-off between dependability (for higher dependability, one might select more variants) and efficiency (for better performance, one might select fewer variants, if few hosts are available to run them in parallel on).

N-Version Programming

In the N-version programming approach, there are two or more *versions* of a component, plus an execution environment that has a *decision algorithm* [8]. All of the versions are executed. Their results are submitted to the decision algorithm, which has the job of producing a result which is believed to be correct (or possibly declaring that it can't find a correct result).

Error detection is via comparing variants; recovery may be forward recovery with a "community error recovery" algorithm (in which failed versions can be fed the correct results of other versions) or compensation.

Avizienis [8] identifies several parameters for this approach: the "cross-check points" to run the decision algorithm at, the "recovery points" (which may be a subset of the cross-check points) where community error recovery is feasible, and the choice of decision algorithm(s) plus the kinds of responses to their possible outcomes. (Examples of outcomes are: a majority of identical responses, a plurality of identical responses, a pair of identical responses, all responses differ.)

N-Modular Redundancy

N-version programming is an extension of the hardware fault tolerance technique of N-modular redundancy [8] (or tri-modular redundancy). One might invert this statement and consider N-modular redundancy a degenerate case of N-version programming in which the software versions are identical copies but run on different hardware; some hardware faults can be tolerated but software faults held in common between the copies would not be.

N Self-Checking Programming

A "self-checking component" contains enough redundancy to self-detect one error: it is composed of either one variant and one acceptance test or else two variants and a comparison algorithm [33]. In the n-self-checking programming approach, a software component is composed of two (or more) of these self-checking components: one is designated as primary, and the other(s) are hot spares running on separate hardware.

If the primary detects a failure in its own result, one of the spares takes over as primary; meanwhile the old primary attempts to recover so that it can become a spare.

4.5 Construction via Connector Transformations

Here I describe how to construct a significant subset of the preceding dependability techniques in terms of connector transformations and compositions of connector transformations; I give the reasoning for selecting the transformations and, where appropriate, briefly rephrase in terms of the transformation parameters given in sec-

	Data translation	Add observer	Add redirect	Add switch	Add parallel	Aggregate	Sessionize	Splice
Acceptance test (various)	*		■				*	
Comparing variants					■			
Roll backward: checkpoint			■					*
Roll forward: community ER		■						*
Compensate: with vote					■			
Compensate: with FEC	■ +							
Remove variant					■			
Degradation: missing push			■			■		*
Degradation: missing pull				■		■		*
Degradation: filter requests			■			■		
Failover: cold				■				
Failover: warm					■		*	
Reroute	■ +					■		
Recovery blocks (RB)			■ +	■				
Primary/shadow pair			■		■			
Standby sparing			■	■				
Distributed RB			■ +		■			
Consensus RB		*	■ +		■			
Retry blocks w/ data diversity	■ +		■				■	
Retry	■ +		■				■	
Self−configuring optimal programming	*This technique is not handled.*							
N−modular redundancy					■			
N−version programming		*			■			
N self−checking programming			■ +		■			

KEY

■ Needs one of this kind of transformation

■ + Needs two (or more) of this transformation

* May use this transformation (zero or more)

Table 4.3: Dependability techniques: connector transformations needed

tions 3.4–3.12. I focus here on connector related issues; specifically I make the assumption that all variants of a component present (to the connector) similar interfaces[1] (or "roles"), and I do not attempt to address the issue of the creation of the variants themselves.

Table 4.3 summarizes the remainder of this chapter. Each row represents a dependability technique that has been defined earlier in the chapter; filled squares show

[1]Note however that the splice transformation may be of use in the case that variants present somewhat different interfaces.

which types of connector transformations are needed, at minimum, to add that dependability technique to a connector. In addition to indicating the coverage of techniques, this table shows that often a *composition of transformations* is used to achieve *one enhancement*. Let's now consider each of these techniques in turn.

4.5.1 Acceptance Tests

Acceptance tests (defined on page 54) are application specific and check the results of a computation; they may also require timing information, past results, the computation's inputs, and intermediate results.

Acceptance test: boundary conditions, rate of change

When only the current result is required (which may be the case for a test of *boundary conditions*), one might use an *add redirect* transformation (another possible alternative, to be used for simple memoryless acceptance tests that can be stated as a function, is to use a *data transformation*). The redirect transformation may also be used in this way when the current result plus a history of past results is required; if timestamps are needed in addition to an internal history (which may be the case for a test of *rate of change*) and are not already present in the data, they may be added with a *data transformation* in addition to the redirect.

Let S (a role) be the source of the questionable transmission; let D (a set of roles) be the original destination(s) of the transmission.

If we choose to use a redirect: Given the redirect form "TrafficDirector $\times$ Role $\times$ TrafficSet $\times$ Connector $\rightarrow$ Connector", the TrafficSet is the set of transmissions from S that are to be checked; the Role represents a component that accepts traffic from TrafficSet (and may keep an internal history of it) and outputs either the received value, indicating acceptance, or an error, indicating non-acceptance; the TrafficDirector redirects traffic (in TrafficSet) from S to Role and always sends Role's response to D.

If we use a redirect and a data transformation (for timestamps), the desired Location of the data transformation is the sender role of the data to which a timestamp is being appended (in this case, S); the Function is to append a timestamp. Either the component added by the redirect transformation (referred to as "Role" above) must remove the appended timestamp when a correct output is relayed from Role to D, or an additional data transformation may be used (with Location at receiver, D, and a

Function of removing the appended timestamp) so that D receives outputs that do not have a timestamp appended.

If we choose to use a data transformation for a simple memoryless acceptance test, given the data transformation form "Function $\times$ Location $\times$ Connector $\rightarrow$ Connector", Function is a function that maps out-of-bounds values to "error" and behaves as the identity function for all other values; Location may be at S or at (each of, if there is more than one) D.

Acceptance test: different accuracy, inverse function

When the current result plus the computation's inputs are required (which is the case for a test that uses *different accuracy* or computes the *inverse function*), we still use an *add redirect* transformation, but it is now necessary to first intercept the inputs as well and be able to associate them with the intercepted result.

In the case that D (above) is issuing requests, to which the outputs from S to D are responses, we can use a *sessionize* transformation to record the inputs (initiating a session for each request) and replay the inputs together with the response when relaying a response to the new Role. If, however, the inputs are coming from another source not accessible from the S–D connector we are engaged in transforming, it may be necessary to modify the component, just as with a certification trail (below).

Acceptance test: certification trail

This technique requires additional information to be communicated to the added role. To obtain the new information will require *modification of the original component* that performs the computation. Assume for the moment that the component has been modified to output this additional information; we would then wish to *redirect* it to the added role. TrafficSet would be exactly this additional information plus the results sent by S; Role then relays only the results (or issues an error instead).

Acceptance test: watchdog timer

Connectors for distributed systems often have a timeout mechanism built in. A connector that lacks a timeout can be augmented with one by adding a "clock" that starts ticking when anything requiring a reply is sent and either stops if a reply is received in time or sends a timeout event to the sender if a reply is not received in time. That is, the "clock" observes communication traffic at the sender's end of the connector and may also participate in the communication. This augmentation uses an *add redirect* transformation: requests are intercepted (and copied to the added role to start a timer) and relayed to the recipient, and replies are either intercepted (and copied to the added role to stop a timer) *or generated* and relayed to the sender. The component represented by the added role is the "clock."

4.5.2 Comparing Variants

This means of error detection (defined on page 55) employs at least two variants and a *decision algorithm* that operates on their results and (as with an acceptance test) returns a result[2] or an error. Let's assume here that, given a set of variants S_v, every member of the set is compatible with the same role interface R_v.

The variants themselves would be added to the connector via a transformation that adds roles. If they are to operate at the same time, we would use the *add parallel replication* transformation. The decision algorithm, then, is a part of the TrafficDirector introduced by this transformation. (See "N-version programming" below.) Specifically, it is the part responsible for *reconciling responses* from the replicated roles.

4.5.3 Backward Recovery (checkpoint/rollback)

In discussing this augmentation (defined on page 56), I will focus on a component S which is to be checkpointed. Unless the checkpointed state is represented entirely as a log of communications (which could be captured without change to S using *sessionize*), this augmentation is not transparent and will require S's interface to include a means of requesting S's current state, so that the checkpoint can be performed, and a means of sending S a saved state, so that the rollback can be performed. The augmentation also requires a *coordinator* to request and replay the checkpoint. Since the coordinator is a new participant in the communication, this indicates a need for a transformation that adds a new role; we use an *add redirect* transformation. Let's call this new role C for "coordinator".

Requests are redirected to C, the coordinator. When C receives a redirected request, C first sends a "get state" command to S's get-state/set-state interface and records the checkpoint. Then C relays the request to S. Eventually S generates a

[2] We expect that the result returned by the decision algorithm will match at least one of the set of results that were its inputs.

response, which is redirected to the coordinator C. If the response is an error, C sends a "set state" command to S with the checkpoint that C had received from S. Finally, C relays S's response (whether it is an error or not) to its originally intended recipient.

Conversely, given a connector with provision for backward recovery via get-state/set-state, to which we are adding a component (variant) that doesn't have a get-state/set-state interface, we can use a *splice* transformation to remove these requests for the added variant. This splice transformation would respond to C's requests with a null checkpoint on behalf of S. This may be reasonable for a stateless component (e.g., a recovery block variant that isn't guaranteed to be run every time has to be stateless) but would otherwise leave the component in an uncorrected state which will continue to produce incorrect results; it may be preferable instead to remove such a component from participation.

4.5.4 Forward Recovery (community error recovery)

Community error recovery (defined on page 56), in which a correct value (from a fellow variant) is relayed to a component that generated an incorrect value, requires that the component has an interface where such corrections can be submitted. (The component is also assumed to be able to make use of the correct value to move forward to a consistent state.) One could make use of an *add observer* transformation to achieve this; correct results of the majority variants are copied to the correction submission interface of the minority variants.

Conversely, if we have a connector with provision for issuing corrections, and we are adding a component (variant) that doesn't have an interface for accepting corrections, we can use an *add splice* transformation to remove the corrections that would be received by that variant (which may then, of course, continue to generate incorrect results).

4.5.5 Compensation

In the case of multiple variants (via added role), compensation (defined on page 57) would be placed in the TrafficDirector, in the part responsible for *reconciling responses* from the replicated roles (see Comparing Variants, above).

Compensation techniques may also be used in transmission to mask errors introduced in the communication channel, as with detection/correction codes. This augmentation adds redundancy (e.g., a checksum) within a transmission. The transmission is restored to its original (nonredundant) format at the receiver, and, in the process, an error may be masked or detected. These codes can be added with a *data translation* transformation.

We require two applications of the transformation. Given a data translation form "Function $\times$ Location $\times$ Connector $\rightarrow$ Connector", in the first application, Location is at the sender and Function is a function from "data" to "data plus detection code"; in the second application, Location is at the receiver and Function is a function from "data plus detection code" to "data" or "error."

4.5.6 Fault Handling via Variant Removal

As with compensation, this may be placed in the TrafficDirector: the TrafficDirector can cease to issue requests to the faulty variant.

4.5.7 Fault Handling via Degradation

Constructing a system that can gracefully degrade (defined on page 57) is likely to affect the component design as well as the connectors, but connectors can help to support the degradation process locally.

For example, the information a connector provides to a component C may be essential or nonessential. When the originator of nonessential information fails, in a gracefully degrading system C should continue to operate. If C was not designed and implemented with the expectation that nonessential inputs may suddenly "go away", and C can only continue to operate if default values for these inputs are provided, we can modify the connector to provide defaults for nonessential information so that C will continue to function. The connector transformation appropriate for this enhancement will depend on whether the information ordinarily arrives as a "push" or a "pull"; in the former case, this enhancement is similar to a watchdog timer (but instead of generating timeout errors, it generates a default value), and in the latter case, it may be considered a degenerate form of failover (discussed below) with a trivial backup component that simply provides default values. An alternative approach is to use an aggregate transformation to combine a connector representing the ideal configuration (all inputs available, all types of requests serviced) with a connector representing the degraded configuration (some inputs unavailable, some types of requests dropped); an *add redirect* or *splice* transformation may be used to construct the latter connector, as described below.

Now suppose instead that we have a component S that provides essential and nonessential services. If failures of other components, or other abnormal circumstances, suddenly result in an unreasonably high load for S, we may wish to modify a connector to temporarily filter out requests for nonessential services (with the assumption that the requesting components are able to continue operating without the nonessential inputs provided by S). We can construct this modification from an aggregate transformation that combines an unmodified connector and a "filtering" connector. To create the filtering connector, we may use an *add redirect* transformation (like the mini-cache described below in Retry, it should be co-located with the receiving component S) or a *splice* transformation. In the case of an add-redirect, incoming requests are directed to the new role, which determines whether a request is essential (and relays it to S) or nonessential (and rejects it). We may use a splice transformation instead if the communication protocol itself is reduced (e.g. entire classes of events are filtered).

4.5.8 Failover (a piece of Recovery Blocks)

Several of the recovery-block-based techniques listed earlier (standby sparing, PSP, recovery blocks, distributed RB, consensus RB, defined on pages 58 through 61) have some building blocks in common: an error detection technique, a backward recovery technique, and a piece that I will refer to here as "failover."

The "failover" augmentation adds a replacement component that is on standby (it may be observing traffic, but it is not actively participating in it) until the failure of the component that it is intended to cover for. Because it adds a potential participant to the communication, we use a transformation that adds a role. More specifically, if the original component and the replacement are taking turns (rather than being in simultaneous operation as with n-version programming), the *add switch* transformation would be the most appropriate. With only an add switch transformation, this results in a cold failover (the added role doesn't see any traffic until it is activated, on the failure of the original role's component). A warm failover can be achieved either by composing the add-switch with an *add observer* transformation (so that prior to switching, the added role functions as an observer) or by using an *add parallel* transformation instead of the add-switch (and requiring the TrafficDirector to behave essentially as a switch). For some applications a cold failover might also be warmed up by replaying a log of requests received by the primary component. Here we'll consider a cold failover:

Let R be the role representing the original component. Given the switch form "TrafficDirector $\times$ Role $\times$ TrafficSet $\times$ Connector $\rightarrow$ Connector", Role represents the replacement component and must be compatible with R; TrafficSet is everything sent to R; TrafficDirector directs transmissions in TrafficSet normally until the failure of R is suspected (by some detection technique used in conjunction with Failover), whereupon requests sent to R are redirected to Role (and Role's responses are sent to the originator of the request, as R would have done).

4.5.9 Reroute (the dual of Failover)

If the possible failure of an existing communication channel is anticipated at design time, a means of temporarily rerouting the communication may be incorporated. This technique is the dual of "failover", in the sense that it provides a backup *connector* should the failure of the primary connector be detected. Abstractly, the end result is a complex connector that offers an alternative between two connectors (the primary and the rerouted backup), and we use the *aggregate* transformation. Note that if the backup route is an indirect path that makes use of existing segments that ordinarily carry other traffic, it may be necessary to add a *data translation* transformation to tag the rerouted traffic at the sender and to remove this tag at the ultimate recipient.

Given the aggregate form "Selector $\times$ SelectionPoints $\times$ Connector $\ldots \times$ Connector $\rightarrow$ Connector", the Selector will select the primary connector unless informed (e.g., by receiving an error from an augmentation such as timeout or retry) that the primary connector has failed. The precise SelectionPoints are dependent upon the protocol of the original connectors, but should include a point at initialization and a point sometime after any failure notification; if recovery notifications are possible, it should include a point sometime after them; otherwise, it may include a point at which recovery is assumed so that the primary will be tried again (as a means of detecting recovery). One Connector parameter is the primary connector and the other is the rerouted connector; the roles of these connectors should match (that is, the connectors should be interchangeable from the perspective of the communicating components).

4.5.10 Recovery Blocks

The recovery block approach was defined on page 58. We compose the transformations used for a cold failover, a backward recovery (if supported by the variants), and an acceptance test, in that order: *add switch, add redirect, add redirect.*

A request is first directed to the checkpoint coordinator (via a redirect transformation) and thence to the primary variant. The result is directed first to an acceptance test (via a second redirect transformation) and then to the checkpoint coordinator (via the first redirect transformation), which will ask the primary to roll itself back if the acceptance test indicates an error. The coordinator then in turn relays the response (either a correct response or an error), which the switch transformation uses to trigger a switchover to the next variant in the case of an error.

4.5.11 Primary/Shadow Pair

The PSP approach was defined on page 60. We compose the transformations used for a warm failover and an acceptance test. This is similar to distributed RB (below) with the omission of the backward recovery: we leave out the first *add redirect* transformation.

4.5.12 Standby Sparing

The standby sparing approach was defined on page 60. We compose the transformations used for a cold failover and an acceptance test. This is similar to recovery blocks (above) with the omission of the backward recovery: we leave out the first *add redirect* transformation.

4.5.13 Distributed Recovery Blocks

The distributed recovery block approach was defined on page 60. We compose the transformations used for a warm failover, a backward recovery, and an acceptance test, in that order: *add parallel, add redirect, add redirect.* The response reconciliation of the TrafficDirector must have an ordering of the variants, and selects the first one of the responses (according to that ordering) that is not an error. Note that in recovery blocks (above) this ordering was implicit in the structure of the switch transformation.

4.5.14 Consensus Recovery Blocks

The consensus recovery blocks approach (defined on page 61) is similar to n-version programming (see below), with the addition of acceptance tests to be employed if no consensus is present. The simplest form of this technique to construct with connector transformations is one which first filters the responses through the acceptance test(s) and then submits their results to the response reconciliation of the TrafficDirector. We use *add parallel*, plus one *add redirect* per variant for the acceptance tests.

4.5.15 Retry Blocks with Data Diversity

This technique (defined on page 61) is a variation on recovery blocks in which there is one variant but multiple data re-expression algorithms, which are used to modify the data before a retransmission. To the connector transformations that are used to achieve Retry (below), we will add a *data translation* for each data re-expression algorithm. We may also add backward recovery for the receiving component (as in recovery blocks above), if the receiving component has state and supports backward recovery.

4.5.16 Retry

Retry (defined on page 61) is used in conjunction with a detection technique (such as an acceptance test or an error detection code); when an error is detected, this triggers a request to resend the corrupt or missing data. The augmentation adds state to the connector, since it must remember what to re-request (until the data arrives without an error being detected). We can achieve this with a *sessionize* transformation. If an imprecise detection technique is used, multiple copies of a transmission may result, which in some applications will be undesirable. For example, if the transmission is a request for a non-idempotent service, we must also use a *data translation* transformation at the requester (before the request is recorded), to add a means of detecting duplicate requests, and an *add redirect* transformation, to add a small cache at the non-idempotent service provider.

Let S be the sender of a transmission; let R be the receiver. Given the form "Recorder $\times$ Replayer $\times$ Connector $\rightarrow$ Connector", the Recorder records the transmission when S sends it; the policy for erasure may vary. The Replayer, when triggered, resends the request; the trigger may be an error (including the timeout of an expected response or acknowledgement) or an explicit "resend please." What about duplicate filtering? Given a data translation form "Function $\times$ Location $\times$ Connector $\rightarrow$ Connector" and a redirect form "TrafficDirector $\times$ Role $\times$ TrafficSet $\times$ Connector $\rightarrow$ Connector", Location is at S, Function appends a unique identifier to the request, TrafficSet is the set of requests from S and responses from R, and TrafficDirector redirects requests and responses to Role; Role, a cache, forwards all traffic to its original destination, unless the unique id of a request indicates that it is a duplicate of a recently seen request/response pair, in which case the original response will be repeated to the sender of the duplicate. Note that, in implementation, the mini-cache represented by Role should ideally be co-located with the receiver represented by R, with a reliable channel (e.g., procedure call) between the two.

4.5.17 Send N Times

This technique is a degenerate form of "retry", which used a *data translation* to add a means of identifying duplicates, a *sessionize* transformation to record and resend a transmission, and an *add redirect* transformation to identify and resolve duplicates. The primary differences in this form are the Replayer in the sessionize form and the function of the Role in the redirect form. The new Replayer is triggered immediately, by the outgoing transmission (rather than by an error or an explicit request for retransmission), and replays it $n-1$ times. The Role collects and reconciles duplicates and, for each set of (up to) n duplicates, sends a single transmission to the final destination (with which, as in retry/resend, the Role's component should be co-located in the deployed implementation).

4.5.18 Self-Configuring Optimal Programming

SCOP was defined on page 62. The degree of dynamism supported by the set of connector transformations as described and implemented in this thesis is limited (a tradeoff which, however, facilitates advance analysis of the connector's behavior as in Chapter 5); it would be possible to construct a poor man's SCOP in which configurations are enumerated within the structure of the connector, but to achieve maximum flexibility may require greater support for runtime modification of the system.

4.5.19 N-Modular Redundancy

Structurally, this enhancement (defined on page 63) adds roles to a connector. The new roles serve a purpose similar to an existing role. They operate at the same time as the existing role. Therefore we use the *add parallel replication* transformation.

Let R be the existing role that is to be replicated. Given the parallel replication form "TrafficDirector $\times$ Role $\times$ TrafficSet $\times$ Connector $\rightarrow$ Connector", the TrafficSet is the set of requests issued by others to R; the Role must present an interface compatible with R; TrafficDirector must copy any request in TrafficSet to Role and must compare responses from R and Role, either returning one apparently-correct response or indicating a detected error.

4.5.20 N-Version Programming

From the connector perspective, this technique (defined on page 62) is a variation on n-modular redundancy in which the collection of identical roles happens to represent a collection of differently-implemented component variants. We still use an *add parallel* transformation to add these roles. In addition, we may employ forward recovery (if the component variants support it) via *add observer* transformations.

4.5.21 N Self-Checking Programming

This technique was defined on page 63. Each self-checking variant is a variant, plus an acceptance test, plus backward recovery (thus we use two *add redirect* transformations for each variant); these self-checking variants are combined into a self-checking "component" via the *add parallel* transformation. This is similar in structure to my "filter through acceptance tests, then vote" construction in Consensus Recovery Blocks (section 4.5.14).

4.6 Summary

In Chapter 3, I exhibited a small set of basic structural transformations that can be applied compositionally to simple connectors. Now I have illustrated how such compositions can produce a wide variety of useful complex connectors in the domain of dependability enhancements.

I have shown here that it is possible to select a set of transformations that is manageably small, but still sufficiently powerful to produce augmentations corresponding to well-known useful dependability enhancing techniques when only a few, sometimes only one or two, of the transformations are composed together. That the transformations are generic in terms of extrafunctional enhancement is suggested by the examples of use sketched in sections 3.4–3.12; the more detailed examples in this chapter, which were summarized in Table 4.3, show the degree of coverage provided by connector transformations within the specific extrafunctional domain of dependability. The transformations' applicability to diverse connector types will be demonstrated in succeeding chapters.

So far I have described connector transformations in abstract, yet informal, terms. In Chapter 5 I move away from informality to give a means of formally characterizing the connector transformations in terms of their effect on the connector protocol. In Chapter 6 I move away from abstraction to demonstrate that it is possible to generate implementations of connector transformations and thus, with the assistance of automated tools, create complex connectors.

Chapter 5

Semantics of Connector Transformations

5.1 Introduction

In this chapter, I give a formal basis for connector transformations. The motivation to provide some formal means of understanding and reasoning about connector transformations is largely practical: formalism can provide useful benefits to those who use connector transformations.

Why is a formal description desirable? A good formal description enables more accurate understanding and explanation of the "meaning" of connector transformations than either an English language description or a reference implementation of a transformation tool. It provides a means to describe transformations in a way that is base-connector neutral (i.e., applicable to multiple connector types). In addition, with a formal description it becomes possible to perform analyses that answer important questions about the effect of a particular connector transformation. Such questions include:

- Correctness: Does the transformation do what it claims to?
- Soundness: Having introduced a transformation (or sequence of transformations), does the resulting connector still work, or does the transformation introduce new deadlocks, failure modes or race conditions?
- Transparency: Does a transformation change the interface of the communicating parties? Since the goal of transformations is to avoid directly modifying the

components in a system, transparency is often an important feature to verify.
- Compositionality: What are the compositional and algebraic properties of a set of transformations? This includes issues such as commutativity (can the ordering of two transformations be exchanged?), inverses (does one transformation undo the effects of another?), idempotence (does it matter if the same transformation is applied twice?), and other more specific properties of a composition of several transformations.

In the remainder of the chapter, I present a formal approach that will answer these questions. The key idea is to describe connectors as structured protocols and to use this as a basis for describing connector transformations as transformations on these protocols. After describing the intended audience, I begin with a gentle introduction to the process algebra syntax that will be used in the remainder of the chapter. I explain the structure of a connector and a connector transformation as I have chosen to describe them in this formalism. This is followed by a concrete example, including (in section 5.3.4) how to perform some of the checks listed above (specifically, whether the result is sound or deadlocks, whether wrappers are transparent to the caller role or have changed the interface, whether error-masking has been successful from the point of view of a role, and whether the role is able to make progress). A catalog of patterns for describing connector transformations as protocol transformations is given. I have chosen to model connectors using Finite State Processes (FSP) notation, a means of describing Labeled Transition Systems which can be verified by the Labeled Transition System Analyzer (LTSA) model-checking tool [36]; an optional-reading section on the differences between FSP and Wright appears before the conclusion of the chapter.

5.1.1 Goals and Scope

It is important to be clear about the primary goals of the formal part of the thesis work. In particular the formalism should have two qualities. First, it should be sufficiently expressive to describe or analyze interesting aspects of connectors. Second, it should suit the needs of my target audience, which includes those who will select and use connector transformations in designing and implementing software systems, as well as those who create connector transformations. These tasks require differing levels of expertise. This motivates the use of an approach with reusable parts and a formal notation that supports useful analyses but is no more complex than necessary.

At the level requiring least expertise, there are people who will apply sets of exist-

	Concept	FSP Artifact	Created by
Most general	"Generic" connector transformation	FSP template	(Section 5.4) Formalism expert
...	Domain-specific transformation	Parameterized specification	Domain-specific connector expert
Most specific	Augmented connector	Specification of augmented conn.	System integrator

Table 5.1: Levels of Artifacts

ing connector transformations to generate new connectors. (As observed in Chapter 1, their skills are more likely to be in the area of component integration than in the gritty details of connector implementation.) They make use of transformation specifications (and analyses) and need to know enough of the formalism to follow the meaning of a specification and to fill in the parameters of a parameterized specification. At the next level, there are people who will create specifications for new domain-specific connector transformations. These people make use of general patterns (or templates), such as those given in section 5.4, to assist in the writing of the connector transformation specifications. Finally, new templates themselves may in turn be written by people with greater experience in the formalism. These levels of expertise, and the corresponding artifacts, are summarized in Table 5.1.

I am interested in providing a formalism that provides some leverage for the people who would *use* connector transformations (that is, the lowest and least-expert level of Table 5.1). I also wish to retain sufficient expressiveness to describe and analyze interesting connector transformations of "real world" complexity; however, it is not essential to have a complete, elegant calculus of transformations. Rather, it is more important to be able to use formal descriptions of connector transformations to perform analyses that help to indicate whether a particular transformation is a reasonable match for system requirements, or is inappropriate. This is not to say that one cannot demonstrate useful formal properties (such will be shown later in this chapter); formal relationships, algebraic properties, and the use of tools to explore these things, are still relevant, but are not the primary focus.

I focus specifically on the *protocol* aspect of connectors; this aspect facilitates some types of analyses that are likely to be desirable to my target audience[1]. What

[1] Section 5.3.4 and section 7.7 discuss the means of performing such analyses and compare com-

are some analyses that may be desirable to users of connector transformations? As indicated earlier, in the general case, it would be helpful for analyses to be able to indicate potential problems related to the soundness and the transparency of a transformation, as well as issues relating to the composition of two or more transformations. Analysis of connector protocols can detect deadlocks introduced by an unsound transformation and can determine whether a connector's interfaces are preserved by a transparent transformation (or the extent to which the interfaces are modified by a non-transparent transformation), as well as answering a number of compositionality issues, such as whether two non-commutative transformations are correctly ordered. In addition, as will be seen in later examples, it is possible to formulate domain-specific[2] analyses that can help to show whether a transformation is achieving its purpose (e.g., to show that a reliability-enhancing transformation to a client-server connector successfully masks transient errors from the client). Benefits and drawbacks of focusing on the *protocol* aspect of connectors are discussed further in section 8.2.

5.2 Overview of the Approach

This section will, first, give the reader sufficient background in the syntax of a specific process algebra to understand the remainder of the chapter; second, explain how I have used this process algebra as a basis for describing the protocols of software connectors; and third, explain how I have chosen to describe connector transformations in this context.

5.2.1 Introduction to FSP Process Algebra

Given the decision to focus on protocols as an aspect of connector transformations, it is natural to build on past work in this area that uses process algebras (such as CSP

mutability predictions to implementation to determine whether the prediction was accurate.

[2]The "domain" in *domain-specific* refers to the kind of extrafunctional properties that a transformation might be chosen to enhance. For example, a domain-specific transformation's purpose may be "adding encryption", which one might call an enhancement in the *security* domain; or, again, the intention of a domain-specific transformation may be to increase redundancy in the transmission (a *reliability*-domain enhancement), and an associated domain-specific analysis might show that some class of errors has now been, in theory, eliminated.

and CCS [26, 41]) which have proven useful for describing and analyzing protocols. Process algebras provide a way to talk about patterns of events and are supported by a number of useful analysis tools.

The specific process algebra chosen here is FSP. Other process algebras could have worked as well: I chose FSP because its notation and tool support was designed to be simpler to use than other process algebras[3], and because it provides a useful set of analyses, such as whether a connector protocol will deadlock or whether a safety or liveness property is violated. (Effects of the choice of a simpler notation with fewer operators versus notations with greater expressiveness are discussed in greater detail in section 8.1.2.)

A quick reference for some FSP operators is given in Table 5.2; for a comprehensive reference, see [36]. Processes describe actions (which I will refer to as *events*) that occur in sequence and choices between event sequences. An FSP process can be converted to a Labelled Transition System (state machine). Each process has an *alphabet* of the events that it is aware of (and either engages in or refuses to engage in). When composed in parallel, processes synchronize on *shared* events: if processes P and Q are composed in parallel as $P \parallel Q$, events that are in the alphabet of only one of the two processes can occur independently of the other process, but an event that is in both processes' alphabets cannot occur until both processes can engage in it.

To illustrate with a simple example, processes A and B, operating in parallel as the composite process C, will synchronize on their shared event b (but do not synchronize on non-shared events a and c). An execution trace of C will begin with a, then b, followed by c, a or a, c, then b, followed by c, a or a, c, then b, and so on.

$\mathsf{A} = (\mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{A}).$
$\mathsf{B} = (\mathsf{b} \rightarrow \mathsf{c} \rightarrow \mathsf{B}).$
$\parallel\mathsf{C} = (\mathsf{A} \parallel \mathsf{B}).$

$P = (a\!\rightarrow Q)$ is a process that engages in a and then behaves as process Q, where a is an "action label" representing some event. $P = (a\!\rightarrow Q|\ b\!\rightarrow R)$ engages in either a or b, and, subsequently, behaves as the corresponding process Q (if a was chosen) or R (if b). $P = (a\!\rightarrow Q|\ a\!\rightarrow R)$ is a legal process and represents a followed by a nondeterministic choice between Q and R. There is otherwise no distinction between

[3]FSP was created as a teaching vehicle to help college students understand concurrency issues.

a → P	Action Prefix (a is an event, P is a process)
P = (a → P).	process definition
P = (a → Q), Q = (b → P).	Q is a locally defined process (scope is within P's definition)
a → P | b → Q	Choice
P ∥ Q	Parallel Composition
∥C = (A ∥ B).	composite process definition
when (n<T) a → P	Guarded Action
P(T=3,M=2) = Q[0], ...	parameterized process
Q[n:0..3] = (when (n<3) ...).	indexed process
Q[n:0..T] = (when (n<T) ...).	indexed process using a parameter
a[i:1..2]	indexed event
a[i:1..M]	indexed event using a parameter
set ExampleSet = {*x*,*y*,*z*}	set definition
a[i:ExampleSet]	indexed event using a defined set
a.b.c	event constructed from three labels a, b, c
one:P	Process Labelling (with label "one")
{one,two}::P	Process Labelling (with set of labels "one", "two")
P/{new/old}	Relabelling
P\{hidden}	Hiding
P+ {a,b,c}	Alphabet Extension
END	predefined process (denotes acceptable termination)
STOP	predefined process (denotes bad termination)
ERROR	predefined process (denotes bad termination)
// This is a comment.	Comment

Table 5.2: FSP Quick Reference

internal and external choice (as there is in CSP), and no distinction between input (or observed) events and output (or initiated) events. Choice may be restricted by a guard; in $P =$ (when B $a\rightarrow Q \mid b\rightarrow R$), the process P cannot engage in a if the predicate B is not true. An advanced example follows to illustrate this and other syntax that will be used in the remainder of the chapter.

Customer(T=3,M=2) = Buy[0],
Buy[n:0..T] = (when (n<T) dollar[i:1..M] → (reject → Buy[n+1] | accept → END)

```
                | when (n==T) kick → END)/{pay/dollar[i:1..M]}.
Vending = (pay → accept → Vending | pay → reject → Vending
             | wait → Vending)\{wait}.
// ||SingleVend = (Customer/{curse/kick} || Vending).
||MultiVend = (one:Customer || two:Customer || {one,two}::Vending).
```

The example above represents a customer and a vending machine; the customer has M limp dollar bills (which are accepted or rejected according to the whim of the vending machine) and the patience for T attempts. Customer(T=3,M=2) is a *parameterized* process; the parameter T is given the value 3 within the scope of the Customer process definition, and M is given the value 2. Buy[n:0..T] is an *indexed* local process; within the scope of the Buy process definition, the index n may take a value from a specified set of action labels or from a specified finite range (here, 0 to T). An action label may also be indexed, such as dollar[i:1..M] which represents, purely for purposes of illustration, an arbitrary selection from the customer's M available dollars; the value of i can be referenced again within the scope of the same *choice element* (in this example, Buy's reject event is in scope, but kick event is out of scope). Buy has two guarded choice elements. When less than T attempts have been made, the process may engage in an indexed dollar event, followed by either reject (and a return to the start of the Buy process, with the counter incremented) or accept (followed by END, a special process that represents successful (non-erroneous) termination). When T attempts have been made, the process engages in a kick event followed by END. At the end of the process definition for Customer and its local process Buy, we see the *relabelling* operator /{pay/dollar[i:1..M]} which replaces all internal dollar events (regardless of index) with pay events, purely for purposes of illustration.

The Vending process engages in pay events (followed nondeterministically by accept or reject), or it may engage in a *hidden* event wait. The hiding operator \{wait}, at the end of the Vending process definition, replaces each hidden event with a special event tau. Since pay is in the alphabet of both Vending and the (relabelled) Customer process, neither of them can engage in pay unless and until the other is prepared to engage in pay also. Since wait is hidden in Vending, even if wait appeared in another process's alphabet, Vending would not have to synchronize on wait.

A comment ("//") follows. The composite process SingleVend combines one Customer and one Vending process. SingleVend illustrates another place that the hiding operator and the relabelling operator can be used (here, the kick event in Customer is replaced with the less violent curse; it makes no difference in this example, since Vending does not use either event), but is otherwise not particularly novel. Suppose instead that we wanted to model more than one Customer? Let's discard (or comment out) SingleVend and consider MultiVend.

The composite process MultiVend has two copies of the Customer process, and one copy of the Vending process. The Customers will operate independently of one another (their alphabets have been made disjoint through process labelling) and the Vending process will synchronize with either. Note that once Vending engages in a pay event, it will not accept another pay event before responding with accept or reject; Vending restricts the interleaving of the Customers' behavior[4]. The *process labelling* operator one:Customer prefixes every event in its copy of Customer with the label one (pay becomes one.pay, and so on). A similar operator {one,two}::Vending prefixes every event in Vending with a *set* of labels; pay becomes {one,two}.pay (so that where a pay event had been acceptable, either of one.pay or two.pay will now be accepted in its place), and so on.

Multiplicity of Customers could also be obtained by indexing with numbers, and the composite process itself can be parameterized:

||MegaVend(N=5) = ([i:1..N]:Customer || {[i:1..N]}::Vending).

FSP supports "safety" and "progress" analyses that require a little additional explanation. Consider again the simple A B C example (repeated here for convenience):

A = (a → b → A).
B = (b → c → B).
||C = (A || B).

Suppose we have a requirement that, in the composite process, a must always be followed by b, and that c must never occur. We can state this as a *safety property*,

[4]What if we were not sure that Vending would restrict the interleaving, and we wanted to check this perhaps-crucial assumption? Consider the situation in which we replace Vending with a simple process P = (pay → P | accept → P | reject → P). We can write a process Q to check for *interleaved payment events*: Q = ({one,two}.pay → ({one,two}.pay → ERROR | {one,two}.{accept,reject} → Q)). We would add this process Q to the MultiVend composite process. Then if it is possible for two "pay" events to occur without an intervening "accept" or "reject" an error will be indicated. Q will cause transition to the ERROR state if P is used in place of the Vending process, showing that P, unlike Vending, does not restrict the interleaving.

which has two parts: first, a process that constrains event ordering of legal events, and second, a set of illegal events. A safety violation will be detected if either the event ordering constraint is violated or it is possible for an illegal event to occur. Our safety requirement is stated as the property BogusSafety below (note that it must also be added to the definition of C). In this case, the first part of the safety property indicates that events a and b are legal and must appear in that order, and the second part of the safety property indicates that c is in the set of illegal events.

property BogusSafety = (a → b → BogusSafety) + {c}.
A = (a → b → A).
B = (b → c → B).
||C = (A || B || BogusSafety).

We already know that this safety property is violated, because a trace of C's behavior will begin $a, b, c, \ldots$ or $a, b, a, c, \ldots$ — the ordering of legal events a, b is ok, but because c can occur there is a violation.

Suppose instead that we have a *progress* requirement that says, as long as the event c keeps happening, the system is ok. We can state a *progress property*, which simply is a set of events. The system is considered to be making progress when *at least one* of these events occurs infinitely often. (There is a progress violation if it is possible for the system to get stuck in a set of states where *none* of those events can ever happen again.) Our progress requirement is stated as the property DoingOk below. Since c does occur infinitely often, there is no violation.

progress DoingOk = {c}
A = (a → b → A).
B = (b → c → B).
||C = (A || B).

This is enough syntax to understand the remainder of the chapter. Next, conventions for describing connectors in a stylized form of FSP will be introduced.

5.2.2 Describing Connectors in FSP

One could use "plain vanilla" FSP to describe connectors, but it is more convenient to have a structured representation that is tailored to the task of connector description.

$\textbf{Connector}\ \mathsf{ProcedureCall} =$

$\quad \textbf{Role}\ \mathsf{Caller} = \overline{\mathsf{call}} \rightarrow \mathsf{return} \rightarrow \mathsf{Caller}$

$\quad \textbf{Role}\ \mathsf{Definer} = \mathsf{call} \rightarrow \overline{\mathsf{return}} \rightarrow \mathsf{Definer}$

$\quad \textbf{Glue} = \mathsf{Caller.call} \rightarrow \overline{\mathsf{Definer.call}} \rightarrow \mathsf{Glue} \ \square\ \mathsf{Definer.return} \rightarrow \overline{\mathsf{Caller.return}} \rightarrow \mathsf{Glue}$

Figure 5.1: Simple Procedure Call — in Wright

Therefore I have adopted a common approach from software architecture[5] in which connectors are described in terms of interface specifications (of the participants in the communication) plus a specification of the glue (the interactions between the participants). In particular, I have chosen to build on the past work of Wright [3] in which protocol semantics are defined in terms of a process algebra (CSP). Wright has a long history of useful formal specification of connectors, such as the HLA [4] discussed in section 2.2 of Related Work.

Wright's decomposition of connectors into interfaces (roles) and interactions (glue) is advantageous because it enables explicit identification of the communicating parties and their obligations, as well as compatibility checks. Wright allows one to describe entire systems containing components in addition to connectors; I will be considering only connectors.

A connector is defined as a set of processes: there is one process for each interface or "role" of the connector, plus one process for the "glue" that describes how all the roles are bound together. These role processes and the glue process are ultimately placed in parallel with one another to form a composite process representing the connector. A simple Wright example is shown in Figure 5.1. This procedure-call connector has two roles, Caller and Definer, plus the Glue that describes their interaction. The caller initiates a call event and then observes a return event; the definer observes a call event and then initiates a return event. The glue observes the caller's call and passes it to the definer, or observes the definer's return and passes it to the caller (the choice between call and return is *external,* dictated by the glue's environment as is proper for observed events, rather than *internal,* decided upon by the glue itself as is proper for initiated events). Note that the glue is more permissive of event ordering than the roles are: in this case, the roles require that each call must be followed by a

[5]The treatment of connectors as first-class entities, a fundamental concept on which this work builds, is discussed further in Chapter 2, Related Work.

Caller = (call → return → Caller).
Definer = (call → return → Definer).
Glue = (caller.call → definer.call → Glue | definer.return → caller.return → Glue).
||ProcCall = (caller:Caller || definer:Definer || Glue).

Figure 5.2: Simple Procedure Call — in FSP

return before another call can be made, whereas the glue makes no such commitment.

A major strength of formalisms such as Wright is the ability to perform analyses. Standard checks in Wright[6] include: freedom from deadlocks in the connector interfaces (roles); freedom from deadlocks in the connector; initiated events have exactly one initiator; a choice between initiated events is made by the initiator not its environment; parameter substitution; range checks; consistency between component interfaces (ports) and component computation; compatibility between component interfaces and connector interfaces; constraints and consistency of the architectural style; and completeness of attachments between ports and roles. (See [1] for details.)

Here I will take the structure of Wright and use it to describe connectors in FSP. Formally, for a connector with roles $R_1 \ldots R_n$ and glue G, the semantics of the connector is given by Expression 5.1.

$$R_1 \| \ \ldots \| \ R_n \| \ G \tag{5.1}$$

Having seen a simple example in Wright, what would the same connector look like in a Wright-structured form of FSP? Figure 5.2 shows the equivalent procedure-call connector in FSP. Again, the connector has two roles, Caller and Definer. Each engages in a call event followed by a return event. In the composite process ProcCall, the role processes execute concurrently with the Glue process, synchronizing on shared events; here the events in each role process have been prefixed with a label unique to that role (caller or definer), so each role shares events only with the glue (whose events are already qualified with the labels), but not directly with other roles. The glue describes the interaction of the roles: a call event at the Caller role is followed by a call at the Definer role, and a return event at the Definer role is followed by a return at the Caller role. ProcCall can then be checked for deadlock, without needing any

[6]Some of these checks have no equivalent in FSP because it does not explicitly distinguish between initiating and observing an event.

specification of the components whose communication it describes.[7]

The FSP connector is similar in appearance to the Wright example, the chief difference here being that FSP, unlike Wright, does not distinguish between initiated/observed events or between internal/external choice, and that the parallel composition of the processes that make up the connector is explicit in FSP and implicit in the Connector construct of Wright. Implications and additional differences between FSP and Wright are discussed in greater technical detail in section 5.5.

5.2.3 Describing Connector Transformations

So far I've covered a quick overview of FSP syntax and a convention for describing connector protocols in FSP. The key question that remains unanswered is: how can one describe *connector transformations*? Consider a connector transformation in terms of its effect on a connector's protocol: it takes a connector protocol (in the form of Expression 5.1), and produces a new protocol that has, perhaps, been augmented in some way. One would like this new protocol to have the *same form* as Expression 5.1, so that ultimately[8] one can chain together a sequence of transformations, each augmenting the connector that resulted from the previous transformation. By restating these two considerations, we arrive at a high-level description of a connector protocol transformation:

A *connector protocol transformation* takes a connector protocol of the form given in Expression 5.1 and, by adding and modifying processes in that protocol, produces a new connector protocol with the same general form (that is, composed of roles and glue).

In the *most general* case, a connector transformation may be described as a set of functions applied to the roles and glue processes of the original connector, plus a set of k additional role processes. The functions $f_{\ldots}$ are functions from a process to a (perhaps composite) process.

$$f_1(R_1) \| \ \ldots \| \ f_n(R_n) \| \ f_G(G) \| \ R_{n+1} \| \ \ldots \| \ R_{n+k} \tag{5.2}$$

Having presented an expression representing the general case, I will now describe

[7]The protocol of the component interfaces, when available, should be checked for conformance to the role specifications, which are effectively standing in for these future components.

[8]As shown in section 5.3.3.

several common cases that are more specialized and, as a result, simpler.

Some transformations do not add any roles. For example, in a *data translation* transformation, the communicated data is translated to some other format (for example, the transformation causes the data to be compressed or encrypted in transit) but no new role is added. Transformations that do not add roles may be expressed as follows:

$$f_1(R_1) \parallel \ldots \parallel f_n(R_n) \parallel f_G(G) \tag{5.3}$$

Some transformations only add roles, without altering the existing roles (the glue, which coordinates the roles, is still affected):

$$R_1 \parallel \ldots \parallel R_n \parallel f_G(G) \parallel R_{n+1} \parallel \ldots \parallel R_{n+k} \tag{5.4}$$

Many transformations preserve the role interfaces from the original connector, so that the transformed connector can be used without altering the protocol perceived by the communicating components. In this specialized case, f_i is a function that composes a new (perhaps composite) process W_i with an *unaltered* R_i process. Further, f_G may be a function that performs only a renaming of G's events, leaving G otherwise unaltered.

$$(R_1 \parallel W_1) \parallel \ldots \parallel (R_n \parallel W_n) \parallel f_G(G) \ldots \tag{5.5}$$

To understand this case better, let us consider its simplest possible instance in which only one role is affected:

$$(R_1 \parallel W_1) \parallel R_2 \parallel \ldots \parallel R_n \parallel f_G(G) \tag{5.6}$$

The effect in this instance is to *decouple* G and R_1 (which previously synchronized directly with one another) and to insinuate a new process W_1 that mediates between G and R_1. While much more restricted than the general case of transformation seen in the first expression 5.1, the possible results are still quite powerful since W_1, playing the middleman, can intercept, alter, add, or discard any events that would pass between G and R_1. In an implementation this effect would correspond informally to the interposition of wrapper-like code that is insinuated "between" the component and the connector without substantial alteration to the implementation of either. W_1 can be thought of as a *wrapper process.*

How is the decoupling of G and R_1 achieved? Let $A_{R,G} = \alpha(R) \cap \alpha(G)$ be the set of events shared by a role process R and a glue process G. Decoupling can be achieved by renaming these events in *one* of the two processes, ensuring that the two processes no longer synchronize directly. The new process W is placed in parallel with the role processes and glue process. W translates between the events in $A_{R,G}$ (in the alphabet of R) and their counterparts in the renamed alphabet of G.

We can rewrite expression 5.5 as follows:

$$\begin{aligned}
(R_1 \| W_1) \| \ldots \| (R_n \| W_n) \| f_G(G) &= R_1 \| \ldots \| R_n \| (W_1 \| W_n) \| f_G(G) \\
&= R_1 \| \ldots \| R_n \| (W \| f_G(G)) \\
&= R_1 \| \ldots \| R_n \| G' \qquad (5.7)
\end{aligned}$$

Here G' is a composite process that consists of the event-renamed original glue process G plus the wrapper processes W_i for each affected role. The final form of the expression has the same form as the original expression of a connector in expression 5.1, which is a useful restatement when chaining multiple such transformations together because it makes clear the unchanged nature of the roles.

By describing connector transformations in this way, it's possible to apply one such transformation to multiple specifications of connectors (describing the transformation independently of the connector increases its reusability); it's possible to compose more complex enhancements by chaining together a sequence of transformations, each one in effect augmenting the connector that resulted from the previous transformation; and it's possible to create useful analyses that will help to determine whether a transformation applied in this way produces a sound and desirable result.

The following section will illustrate the specialized cases described above, using examples of transformations applied to a specific connector specification. I will begin with concrete non-parameterized examples; this is not the way that the transformations would ordinarily be written but it provides a more gradual introduction, particularly for those who are unfamiliar with FSP. In section 5.3.5 I will move up in generality, to parameterized versions of the examples; these more closely resemble what would actually be used and serve to illustrate the kinds of parameters that a transformation user might fill in. These parameterized transformations are domain-specific

$$
\begin{aligned}
&\mathsf{Caller} = (\mathsf{call} \to \mathsf{TryCall}),\\
&\qquad \mathsf{TryCall} = (\mathsf{return} \to \mathsf{Caller} \mid \mathsf{err} \to \mathsf{Caller}).\\
&\mathsf{Definer} = (\mathsf{call} \to \mathsf{return} \to \mathsf{Definer} \mid \mathsf{crash} \to \mathsf{END}) \setminus \{\mathsf{crash}\}.\\
&\mathsf{Glue} = (\mathsf{caller.call} \to \mathsf{TryCall} \mid \mathsf{definer.return} \to \mathsf{caller.return} \to \mathsf{Glue}),\\
&\qquad \mathsf{TryCall} = (\mathsf{definer.call} \to \mathsf{Glue} \mid \mathsf{caller.err} \to \mathsf{Glue}).\\
&\|\mathsf{FaultyRPC} = (\mathsf{caller{:}Caller} \parallel \mathsf{definer{:}Definer} \parallel \mathsf{Glue}).
\end{aligned}
$$

Figure 5.3: Remote Procedure Call with Timeouts

but can be created from non-domain-specific patterns by transformation writers (a task requiring greater understanding than the transformation users, as described in section 5.1.1). In section 5.3.6 I will use these examples to illustrate how to get from a general pattern to a parameterized domain-specific transformation (this *is* the way that transformations would ordinarily be written). Finally, the patterns themselves are described in section 5.4.

5.3 Example

In Figure 5.3 we introduce a simple connector that will be used to illustrate reliability-enhancing augmentations. This unreliable remote procedure call connector occasionally generates errors (represented here by err events) that are visible to the caller.

Why is this connector unreliable? The FaultyRPC connector is subject to timeout errors that can occur whenever the caller is attempting to contact the definer. The timeouts are represented by the glue's choice of the caller.err event. These timeouts may be transient in nature (perhaps due to problems in the communications channel) or may be due to the permanent silent failure of the definer component. Other sources of failure that can be modelled (but are not incorporated in this example for the sake of simplicity) include: timeouts when the definer is replying to the caller; non-transient failures that are due to the destruction of the communications channel (such as being severed by a backhoe); failure of the caller; in addition, eventual recovery of the failed component is not modelled.

The task of a dependability-enhancing transformation for this connector is to hide the errors from the caller: it must be possible, for a caller that refuses to engage in the err event, not merely to avoid deadlock but also to make progress.

$$
\begin{array}{l}
\mathit{Caller} = (\mathit{call} \rightarrow \mathit{TryCall}), \\
\qquad \mathit{TryCall} = (\mathit{return} \rightarrow \mathit{Caller} \mid \mathit{err} \rightarrow \mathit{Caller}). \\
\mathit{Definer} = (\mathit{call} \rightarrow \mathit{return} \rightarrow \mathit{Definer} \mid \mathit{crash} \rightarrow \mathit{END}) \setminus \{\mathit{crash}\}. \\
\mathit{Glue} = (\mathit{caller.call} \rightarrow \mathit{TryCall} \mid \mathit{definer.return} \rightarrow \mathit{caller.return} \rightarrow \mathit{Glue}), \\
\qquad \mathit{TryCall} = (\mathit{definer.call} \rightarrow \mathit{Glue} \mid \mathit{caller.err} \rightarrow \mathit{Glue}). \\
\mathsf{Retry(T{=}3)} = \mathsf{Retry[0]}, \\
\quad \mathsf{Retry[n{:}0..T]} = (\mathsf{caller.call} \rightarrow \mathsf{retry.caller.call} \rightarrow \mathsf{Retry[0]} \\
\qquad\qquad \mid \mathsf{retry.caller.return} \rightarrow \mathsf{caller.return} \rightarrow \mathsf{Retry[0]} \\
\qquad\qquad \mid \mathsf{when}\ (\mathsf{n{<}T})\ \mathsf{retry.caller.err} \rightarrow \mathsf{retry.caller.call} \rightarrow \mathsf{Retry[n{+}1]} \\
\qquad\qquad \mid \mathsf{when}\ (\mathsf{n{==}T})\ \mathsf{retry.caller.err} \rightarrow \mathsf{caller.err} \rightarrow \mathsf{Retry[0]}). \\
\|\mathsf{RetryRPC} = (\mathit{caller{:}Caller} \parallel \mathit{definer{:}Definer} \\
\qquad\qquad \parallel \mathsf{Glue}/\{\mathsf{retry.caller/caller}\} \parallel \mathsf{Retry}).
\end{array}
$$

Figure 5.4: Applying Retry

Two transformations will be applied to this connector: one that hides soft (transient) faults, which will be shown in Figure 5.4, and one that hides the failure of the definer, which will be shown in Figure 5.5. Transient faults can be masked by re-sending the request that had timed out; this technique is common in practice, e.g., retransmission in TCP [45]. To mask the component failure, one possible technique is to provide a more reliable (but perhaps low-performance) backup for the primary definer, to be used when a failure is diagnosed; this technique is a stripped-down instance of a general well-known method for introducing redundancy, which includes recovery blocks and N-version programming [35, 47].

The two transformations will each be applied individually before their composition (as in Figure 5.6) is described. These transformations and their composition will be explained in greater detail in the next three subsections.

5.3.1 Retry

The first transformation is shown in Figure 5.4 as it is applied to the example connector of Figure 5.3. The Caller, Definer, and Glue processes of the original connector are carried over unchanged (for ease of readability, these unchanged portions are shown in italics in the new connector). The transformation has made changes in two locations. First, a new process, Retry, has been added to the connector. Second, the composite process (RetryRPC) is modified as compared to the original (FaultyRPC). Let's look at these changes in greater detail.

The new Retry process has two purposes. First, it separates the caller role from the glue of the original connector, mediating the events that they originally shared so that they are now synchronized only indirectly through Retry (this will make it possible for Retry to intercept and alter these events). Second, Retry intercepts the glue's err events before they reach the caller and instead reissues the caller's request to the glue; either the call returns successfully this time (and a transient error has been masked from the caller); if several such attempts are unsuccessful, Retry gives up its attempt to intervene and passes the err event on to the caller. Let's look at Retry line by line.

Retry(T=3) = Retry[0],

The Retry process is parameterized with T, the maximum number of sequential retry attempts to allow. Here T is given the value 3. Furthermore, the Retry process is indexed internally; here we're initializing the index to 0. (Parameters and indices should be familiar from the vending machine example of section 5.2.1.)

Retry[n:0..T] = (caller.call → retry.caller.call → Retry[0]

The Retry process's index n may range from 0 to T. This index counts the number of attempts made and (as we see in the previous line) is initially 0. On the remainder of this line, we see the first of several choice elements: that is, one of the available top-level choices is to have a caller.call event, followed by a retry.caller.call, followed by Retry with the attempt-counter still set to 0. The effect of this choice element is to relay a caller.call event from the caller to the glue. (Recall that we have prevented the caller and glue from synchronizing directly.)

| retry.caller.return → caller.return → Retry[0]

Here is the second of the four top-level choice elements: it is similar to the previous one. The effect of this choice element is to relay a caller.return event from the glue to the caller. These first two choice elements are quite simple, and help to accomplish the first of the two purposes of the Retry.

| when (n<T) retry.caller.err → retry.caller.call → Retry[n+1]

In the third choice element, there is a guard. This choice is available only if n, the number of attempts made so far, has not reached the threshold T. If the guard is satisfied, then a retry.caller.err will be followed by retry.caller.call, and we will return to the Retry process and *increment* the attempt-counter. The effect of this choice element is to intercept the glue's err event before it reaches the caller and to instead replay the call event to the glue. That is, it retries the call. (Remember also that the attempt-counter will be reset to 0 whenever a call returns successfully.)

| when (n==T) retry.caller.err → caller.err → Retry[0]).

In the final choice element, there is another guard. This choice is available only if n has reached the threshold T. If the guard is satisfied, then a retry.caller.err will be followed by caller.err and the attempt-counter is reset to 0. The effect of this choice element is to relay an err event from the glue to the caller. That is, we have already retried the call as many times as the parameter T allows, and now we must stop trying to conceal the timeout error.

Finally, let us consider the change to the composite process:

||RetryRPC = (... || Glue/{retry.caller/caller} || Retry).

Here, the Retry process is added to the protocol definition, and some of the Glue's events are renamed: any event that begins with caller will have this prefix replaced with retry.caller. As you will recall, this renaming acts to decouple the glue process from the caller process so that Retry has the opportunity to intercept events communicated between them.

The Retry process is actually somewhat stylized and can be generated by performing simple alterations to a "no-op" alphabet-translation wrapper process; the first two choice elements would be carried over unchanged.

Retry illustrates interception and replacement of events without change to the interfaces of the connector. Connector transformations can also enable the addition of new participants (roles) to the communication, as shown in the next transformation example.

5.3.2 Failover

The Failover transformation shown in Figure 5.5 retains the original roles and glue processes of the original connector and introduces a new role which will be used in place of the original Definer when the failure of the latter is suspected. Again let's consider this transformation piece by piece.

The original Caller, Definer, and Glue processes are unchanged and shown in italics.

$$
\begin{aligned}
&\mathit{Caller} = (\mathit{call} \rightarrow \mathit{TryCall}),\\
&\qquad \mathit{TryCall} = (\mathit{return} \rightarrow \mathit{Caller} \mid \mathit{err} \rightarrow \mathit{Caller}).\\
&\mathit{Definer} = (\mathit{call} \rightarrow \mathit{return} \rightarrow \mathit{Definer} \mid \mathit{crash} \rightarrow \mathit{END}) \backslash \{\mathsf{crash}\}.\\
&\mathit{Glue} = (\mathit{caller.call} \rightarrow \mathit{TryCall} \mid \mathit{definer.return} \rightarrow \mathit{caller.return} \rightarrow \mathit{Glue}),\\
&\qquad \mathit{TryCall} = (\mathit{definer.call} \rightarrow \mathit{Glue} \mid \mathit{caller.err} \rightarrow \mathit{Glue}).\\
&\mathsf{Backup} = (\mathsf{call} \rightarrow \mathsf{return} \rightarrow \mathsf{Backup}).\\
&\mathsf{BGlue} = (\mathsf{caller.call} \rightarrow \mathsf{definer.call} \rightarrow \mathsf{BGlue}\\
&\qquad \mid \mathsf{definer.return} \rightarrow \mathsf{caller.return} \rightarrow \mathsf{BGlue}).\\
&\mathsf{Failover} = (\mathsf{caller.call} \rightarrow \mathsf{pri.caller.call} \rightarrow \mathsf{Failover}\\
&\qquad \mid \mathsf{pri.caller.return} \rightarrow \mathsf{caller.return} \rightarrow \mathsf{Failover}\\
&\qquad \mid \mathsf{pri.caller.err} \rightarrow \mathsf{bk.caller.call} \rightarrow \mathsf{ToBk}),\\
&\quad \mathsf{ToBk} = (\mathsf{caller.call} \rightarrow \mathsf{bk.caller.call} \rightarrow \mathsf{ToBk}\\
&\qquad \mid \mathsf{bk.caller.return} \rightarrow \mathsf{caller.return} \rightarrow \mathsf{ToBk}) + \{\mathsf{caller.err}\}.\\
&\|\mathsf{FailoverRPC} = (\mathsf{caller{:}Caller} \parallel \mathsf{pri.definer{:}Definer} \parallel \mathsf{pri{:}Glue}\\
&\quad \parallel \mathsf{bk.definer{:}Backup} \parallel \mathsf{bk{:}BGlue} \parallel \mathsf{Failover}).
\end{aligned}
$$

Figure 5.5: Applying Failover

Next we see a new process Backup.

$\mathsf{Backup} = (\mathsf{call} \rightarrow \mathsf{return} \rightarrow \mathsf{Backup}).$

Backup is simply a Definer that doesn't fail. It represents the interface (role) that will be associated with a reliable-backup component.

$\mathsf{BGlue} = \mathsf{caller.call} \rightarrow \mathsf{definer.call} \rightarrow \mathsf{BGlue}$

$\mid \mathsf{definer.return} \rightarrow \mathsf{caller.return} \rightarrow \mathsf{BGlue}).$

Next we see another new process, BGlue. BGlue is simply a Glue that doesn't fail. (It's possible to model transient and permanent failures of the backup component, but to keep this example simple, both the backup component and the communication channel to it are assumed to be completely reliable.)

$\mathsf{Failover} = (\mathsf{caller.call} \rightarrow \mathsf{pri.caller.call} \rightarrow \mathsf{Failover}$

Finally we see the Failover process which will coordinate the existing Glue and the new processes. The first of several possible choices is for a call event to be passed from the caller to... to what? It's necessary to look ahead briefly to the composite process definition of FailoverRPC: the processes associated with the "primary" definer

(Glue and Definer) are going to be tagged with pri, and the processes associated with the "backup" definer (BGlue and Backup) are going to be tagged with bk. So, here Failover is relaying a call event from the caller to the "glue" of the *primary definer.*

| pri.caller.return $\rightarrow$ caller.return $\rightarrow$ Failover

Having understood the previous line, the effect of this line is obvious: to relay a return event, from the "glue" of the primary definer, to the caller. These two lines are simply mediating between Caller and Glue, which previously synchronized directly but now have been decoupled by renaming.

| pri.caller.err $\rightarrow$ bk.caller.call $\rightarrow$ ToBk),

Aha, now it becomes interesting. An err event from the primary definer's glue is to be intercepted and replaced with a call event sent to the *backup* definer's glue: that is, when an error is detected, the failure of the primary is suspected and the call is replayed to the backup. Then the protocol behaves as the ToBk local process.

ToBk = (caller.call $\rightarrow$ bk.caller.call $\rightarrow$ ToBk

| bk.caller.return $\rightarrow$ caller.return $\rightarrow$ ToBk)+ {caller.err}.

Here is ToBk. It is very similar to the first two lines of Failover: it relays call and return events between two decoupled processes. But instead of mediating between Caller and Glue, it is mediating between Caller and BGlue. That is, the caller requests are redirected to the backup definer, from which the responses are now received.

We add caller.err to the alphabet of Failover, because it is Failover's responsibility to convey events from the glue to the client; the alphabet of Failover must contain all events that were previously shared by the client and the glue, so that the client's ability to engage in those events remains constrained by another process that shares those events.

Finally, as mentioned above, the FailoverRPC composite process adds "pri" and "bk" prefixes to the primary and backup definer processes, both so that they can be differentiated from one another and so that they are decoupled from the Caller process.

The Failover process is essentially a sequential combination of two "no-op" alphabet-translation wrapper processes, plus a "trigger" choice element that switches from one to the other. The first two lines come from the first of the two no-op processes, and the two lines of ToBk come from the second no-op process.

||ReOver = (caller:Caller || pri.definer:Definer
|| pri:(**Glue/{retry.caller/caller}** || **Retry**)
|| bk.definer:Backup || bk:BGlue || Failover).

Figure 5.6: Composing Retry and Failover

5.3.3 Composition

To compose the Retry and Failover wrappers, the reference to the Glue process in FailoverRPC is replaced with the enhanced "glue" of RetryRPC, as in Figure 5.6.

That is, returning to Expression 5.7 again, first we applied Retry to the original connector C, producing a new connector C' with a *composite* glue process $G' = W_{Retry} \| f(G)$. Then, we applied Failover to the new connector C' (treating $W_{Retry} \| f(G)$ as its glue).

5.3.4 Checking Results

Having applied both wrappers to the faulty connector, one would like to know several things: Is the result sound or will it deadlock? Are the wrappers transparent to the caller role or have they changed the interface? Have errors been hidden from the caller role? Does the caller role always make progress or can the system become wedged? I used the LTSA model checker (Labeled Transition System Analyzer [36], available from the authors of FSP), to confirm that the augmented connector is deadlock-free[9] as well as to answer the remaining questions.

To confirm that the caller's interface need not change, one would restate the original caller role as an FSP "safety property." A safety property has two parts: first, a process that constrains event ordering of legal events; second, a set of illegal events. LTSA will show a safety violation and event trace if the legal event ordering is not followed.

To confirm that errors will not reach the caller role, one would write the NoErrors safety property shown in Figure 5.7. Here event ordering is unconstrained and there is one illegal event, caller.err. LTSA will show a safety violation and event trace, if an

[9]LTSA does not require any additional properties to perform this check; one need only push a button.

property NoErrors = STOP + {caller.err}.
progress CallerOk = {caller.return}
||RetryRPC = (caller:Caller || definer:Definer || Glue/{retry.caller/caller} || Retry
|| **NoErrors**).

Figure 5.7: Safety and Progress

illegal event can occur.

To confirm that the caller will *make progress* in the case of a failed definer, one would define the "progress property" CallerOk, a set of events. The system is making progress when at least one of these events occurs infinitely often. If there is a progress violation, LTSA will show an event trace leading to the violation.

It's also possible to determine that these two wrappers are *not commutative*, by checking the connector produced by one ordering against the connector produced by the other ordering, as a safety property; if the safety property is violated, LTSA will show an event trace leading to the violation. Note that it is often also possible to detect noncommutativity simply *by inspection* of the two state machines that are obtained by applying the transformations first in one order, then in the other order. In this particular case the noncommutativity of Retry and Failover is apparent by inspection, as the replacement of the Glue in RetryRPC with the enhanced "glue" of FailoverRPC yields a clearly non-equivalent state machine in LTSA (a state machine with a different minimum number of states is produced!) With a noncommutative pair of transformations, it is important to get the order of application right; fortunately it is fairly intuitive to translate the desired effect in this example ("try x; if that's not enough, try y") into the right order ("wrap with x, then wrap the composite with y").

5.3.5 Parameterization

Entirely hard-wired examples were shown at first as a gentle introduction. However, to achieve any degree of reusability across different connector specifications, more generalized wrapper processes are needed. First, parameterization will be introduced, which enables a process to be tailored to different alphabets. Then more generic patterns for specification of connector transformations will be described in section 5.4.

set C = {*caller*}
set COut = {*call*}
set CIn = {*return*}
set CErr = {*err*}
set L = {*retry*}
set CInit = {*call*}
Retry(T=3) = hide[e:CInit] → Retry[0][e],
  Retry[n:0..T][olde:COut] = ([r:C].[e:COut] → [L].[r].[e] → Retry[0][e]
    | [L].[r:C].[e:CIn] → [r].[e] → Retry[0][olde]
    | when (n<T) [L].[r:C].[e:CErr] → [L].[r].[**olde**] → Retry[n+1][olde]
    | when (n==T) [L].[r:C].[e:CErr] → [r].[e] → Retry[0][olde])\{hide}.
||RetryRPC = (*caller:Caller* || *definer:Definer* || Glue/{[L].[r:C]/[r]} || Retry).

Figure 5.8: Parameterized Retry

Parameterization of the Retry wrapper is shown in Figure 5.8. If we compare this to the unparameterized version (Figure 5.4), the most obvious difference is the replacement of event labels with selections from sets, e.g., [r:C].[e:COut] has taken the place of caller.call.

To apply the wrapper to a connector, we must fill in the italicized regions, defining several global variables: the set C of "caller" roles, the set COut of events that callers may initiate, the events CIn that callers may receive, and a one-element set NewLabel containing the label to tag the glue and the wrapper with. In the Retry process, values of variables are drawn from these sets and are bound within a choice element. So far all seems straightforward; but, in addition, this wrapper must also remember *which* event the caller attempted to transmit, so that *that* specific event can be repeated if necessary. It is necessary to introduce "olde:COut" as a new argument of the local Retry process in order to cache this event.

To make the binding of parameterized values more concrete, consider the event [r:C].[e:COut] in the first line of the local Retry[n:0..T][olde:COut] process. Any value of r drawn from C is acceptable, and any value of e drawn from COut. (Given the single-element definitions of sets C and COut shown in the example, only one possible event can be constructed, caller.call.) The next event in the sequence will prefix the [r].[e] event (bound to caller.call) with a label drawn from the one-element set NewLabel.

The result is retry.caller.call.

Similarly, in the composite process RetryRPC, the relabelling of the Glue should be parameterized to ensure it matches the labels used in the Retry process; the label drawn from NewLabel (which we don't need to refer to again, so there is no need to give it a variable name) will be added to the beginning of each event prefixed with any label in C. The final lines of the figure show the format for the composite process of a wrapped connector that uses this wrapper; the italicized roles should be filled in with the names of the actual roles.

Note the set CInit, used to initialize the cached event. Due to limitations of FSP syntax, a hidden event is used to select an element of CInit to provide the initial value for the second parameter of Retry[0..T][COut] since non-numeric values cannot be expressed directly in this circumstance. This value is not used unless an error is received before the caller has sent an event (which cannot happen in the example connectors shown here).

Parameterization of Failover (Figure 5.9) is similar. Again, it is desirable to cache the event that may be replayed, so that the proper event can be used; here [LBk].[r].call (with the hardwired event call) is used for the sake of brevity, but event caching could instead be introduced in the same manner as the previous figure.

5.3.6 From Patterns to Parameterization

The ongoing example introduced in section 5.3 began with inflexible connector transformations that were applied to a specific connector. This is not the form that an architect would actually have available initially, but as an example it provides a more gentle introduction to the notation than a more generalized form would. From there I worked *upward* in generality to an equivalent transformation that, through parameterization, could be applied to more than one kind of connector. While still domain-specific, this parameterized version is more widely applicable and resembles the form of transformation that an architect would use (filling in the set definitions and parameters to apply it to a desired connector). However, as outlined in section 5.1.1, in the ordinary course of events, one would create the parameterized version by working *downward* from a still more general form: that is, a domain-specific connector expert would use an FSP template representing a generic connector transformation as the basis for a domain-specific parameterized transformation, which architects can

```
set C = {caller}
set COut = {call}
set CIn = {return}
set CErr = {err}
set LPri = {primary}
set LBk = {backup}
Caller = ...
Definer = (call → return → Definer | crash → END) \{crash}.
Glue = (caller.call → TryCall | definer.return → caller.return → Glue),
      TryCall = (definer.call → Glue | caller.err → Glue).
Backup = (call → return → Backup).
BGlue = (caller.call → definer.call → BGlue
           | definer.return → caller.return → BGlue).
Failover = ([r:C].[e:COut] → [LPri].[r].[e] → Failover
             | [LPri].[r:C].[e:CIn] → [r].[e] → Failover
             | [LPri].[r:C].[CErr] → [LBk].[r].call → ToBk),
  ToBk = ([r:C].[e:COut] → [LBk].[r].[e] → ToBk
             | [LBk].[r:C].[e:CIn] → [r].[e] → ToBk).
||FailoverRPC = (caller:Caller || [LPri].definer:Definer || [LPri]:Glue
     || [LBk].definer:Backup || [LBk]:BGlue || Failover).
```

Figure 5.9: Parameterized Failover

then in turn use to create connector-specific instances of the transformation. Before delving into a catalog of these FSP templates in section 5.4, I will briefly illustrate how a domain-specific connector expert might use this catalog to arrive at the by-now-familiar parameterized Failover example.

Suppose that we want to create the Failover transformation from a template. Which template should we choose? The end result of the Failover transformation is to *add a role*, a backup server which duplicates existing functionality; the connector will *switch to* the backup when circumstances suggest that the primary server has failed (that is, when the glue attempts to send an error event to the caller, representing a communication timeout). Therefore we conclude that the Failover transformation is a domain-specific instance of a role-adding transformation; in particular, of the "Add Switch" transformation, for which a template is given in Figure 5.17 (for convenience,

```
// Switch template, from Figure 5.17.
Switch = Case1,
          Case1 = ([r:R].[e:RInitiates] → case1.[r].[e] → Case1
                    | case1.[r:R].[e:RObserves] → [R].[e] → Case1
                    | case1.[r:R].[e:Switch1] → case2.[r].[Start2] → Case2),
          Case2 = ([r:R].[e:RInitiates] → case2.[r].[e] → Case2
                    | case2.[r:R].[e:RObserves] → [r].[e] → Case2
                    | case2.[r:R].[e:Switch2] → case1.[r].[Start1] → Case1).
set R = {caller}
||SwitchingConn2 = ( ...case1.definer:Definer || case2.definer:Definer
                     || case1:Glue || case2:Glue || Switch).

// Failover: replace the following strings in Switch.
// Case1 = Failover, R = C, RInitiates = COut, case1 = [LPri],
// RObserves = CIn, Switch1 = CErr, Case2 = ToBk, case2 = [LBk].
// Note that the Switch2/Start1/Case1 choice element is eliminated.
Failover = ([r:C].[e:COut] → [LPri].[r].[e] → Failover
          | [LPri].[r:C].[e:CIn] → [r].[e] → Failover
          | [LPri].[r:C].[CErr] → [LBk].[r].call → ToBk),
  ToBk = ([r:C].[e:COut] → [LBk].[r].[e] → ToBk
          | [LBk].[r:C].[e:CIn] → [r].[e] → ToBk).
||FailoverRPC = (caller:Caller || [LPri].definer:Definer || [LPri]:Glue
    || [LBk].definer:Backup || [LBk]:BGlue || Failover).
```

Figure 5.10: Making Failover from Switch

relevant parts of Switch and the parameterized Failover are duplicated in Figure 5.10).

This template requires us to define several sets of labels or events, as well as requiring us to choose a location for the Switch wrapper.

First let's consider the wrapper location. Two possible locations are given in the template; it can be placed between the glue and the set of switching roles (in the case of Failover as applied in Figure 5.9, the switching roles are Definer and Backup), or it can be placed on the "other side" of the glue to mediate between the "unaffected" roles (in this case, Caller) and the glue processes. We must choose the second location, because we wish to associate *different glue processes* with Definer and Backup, to represent the assumption that Backup has a more reliable transmission channel than Definer. Our wrapper, then, will mediate between Caller and the two glues Glue and BGlue.

Before considering set definitions, it is useful to note that the name of any set or label in the template can be changed, provided that the change is made throughout and that it does not conflict with a name that is already in use. In general, name changes may be desired to clarify the intent of a domain-specific instance of a template. For example, since we have chosen the second positioning for the wrapper (between the unaffected roles and the glue), the set R in Switch is the set of labels corresponding to the unaffected roles. In the example application of Failover, as stated above, the set of unaffected roles is simply Caller, and its corresponding label is caller. This set R in the Switch template is equivalent to the set C in the original Failover; a designation of "C for caller/client" may be preferred to the more generic "R for role." The name changes for Failover are summarized in Figure 5.10.

Next, we must examine the sets used in the template and determine what (in general terms) they *should* contain in order to produce the desired domain-specific transformation. The chief points to note in creating Failover from Switch are as follows. RObserves and RInitiates are events that are relayed to/from the caller *without* triggering a shift from Definer to Backup. For our purposes in constructing Failover, then, they are the *non-error* events that are shared between the caller and the glue. Switch1 is the set of events that trigger a shift from Definer to Backup (and are not relayed to a role). For these purposes, they are the *error* events, which we wish to intercept before they reach the caller. Start2 represents an event used to prime the pump at a role that we have just switched to. For the case of Failover, this event will simply repeat the call event that had returned an error, but now it will be sent to the backup role instead. This launches us into the local Case2 process. Again, the RObserves and RInitiates events are those that will be relayed to/from the caller without triggering a shift. In fact, for this Failover, we do not want *any* events to trigger a shift from the Backup back to the Definer (recall that, in the Failover example, there was no provision for the Definer role to ever recover from a crash, thus there is no reason ever to switch back to Definer after it has been suspected of crashing). Therefore the set Switch2 is, for this Failover, an *empty set* and in fact we can eliminate the choice element that represents the Switch2-triggered transition.

5.3.7 Assumptions about Base Connectors

In this approach certain assumptions are made about the base connector to which the transformations are being applied.

It is assumed that, unless the transformation is specifically intended to overcome deadlock-inducing protocol mismatch, the base connector is sound (does not deadlock). For other transformations, an unsound base connector will naturally result in an unsound new connector.

It is assumed that, if a transformation is designed to *preserve* (rather than *add*) some desirable property p, this property holds for the original base connector. For example, consider the property of "exactly once" delivery (e.g. a connector between a client and server may promise that any request from the client is delivered to the server without being either lost or visibly duplicated, and similarly for responses from the server), a transformation that preserves this property (e.g. a data translation such as on-the-fly compression), and a base connector in which the property is *not* present (e.g. a connector with either no delivery guarantee or "at least once" delivery). The resulting connector is unlikely to have "exactly once" semantics, though the transformation may otherwise have been successful. In the worst case, if the domain-specific transformation itself makes a strong assumption about the presence of this property, the resulting connector may not even be sound (the connector may deadlock, by expecting a specific missing event) or may not be correct (the transformation may not have the predicted effect because this effect required the missing property).

5.3.8 Useful Qualities of the Approach

The approach introduced in this chapter both isolates the transformation from the rest of the original connector and produces a result whose form can be made to correspond to my original form of a connector, yielding several useful properties.

These transformation specifications are readily composable. This was demonstrated specifically for the case of retry plus failover. In the general case, $W \| f(G)$ is treated as a new *composite* glue, and we apply a new wrapper process W' to the W-wrapped connector:

$$R_1 \| \ \ldots \| \ R_n \| \ W' \| \ f'(W \| \ f(G)) \tag{5.8}$$

Traceability is facilitated by the decompositional structure of the transformed specification, which separates the effects of a sequence of changes to the protocol so that a problem detected by a model checker can more easily be traced back to the change responsible for introducing the problem. Pinpointing the source of such a problem would be more difficult with a monolithic specification of the protocol. Furthermore, the structure in the specification is not arbitrary but corresponds readily to the structure of the implementation, facilitating traceability of a problem detected in one of several wrapper processes by a model checker to the one of several connector transformations in the implementation that would contain a corresponding bug. This correspondence between wrapper processes and implementation could in principle be enforced by implementation generation tools (for more on this subject, see section 9.3.3).

Parameterization techniques enable the construction of *reusable* connector transformation specifications, applicable to a range of connector specifications. Generalization can be taken further, describing patterns or templates for kinds of transformations in terms of their actions (redirect, record/replay, insert, replace, or discard) on events. The next section deals with such templates.

5.4 Catalog

This section presents a number of generalized patterns for connector transformations; the classification of transformations described here should be familiar from Chapter 3. These patterns can be used to create domain-specific[10] parameterized transformations such as the examples seen earlier.

In outline, this section covers Data Translation, role addition (the Add Observer, Add Switch, Add Redirect, and Add Parallel transformations), Aggregate, Sessionize, and Splice (specifically expanding/compacting a sequence of events, and insertion/removal of an unshared event). The "Add Switch" transformation will be somewhat familiar from the Failover example introduced earlier in this chapter (beginning in section 5.3.2); similarly Sessionize, which saves and replays an event within a "session", may be familiar from the Retry example (introduced in section 5.3.1).

[10]Recall that the "domain" in *domain-specific* here refers to the kind of extrafunctional properties that a transformation might be intended to enhance; e.g., security, reliability.

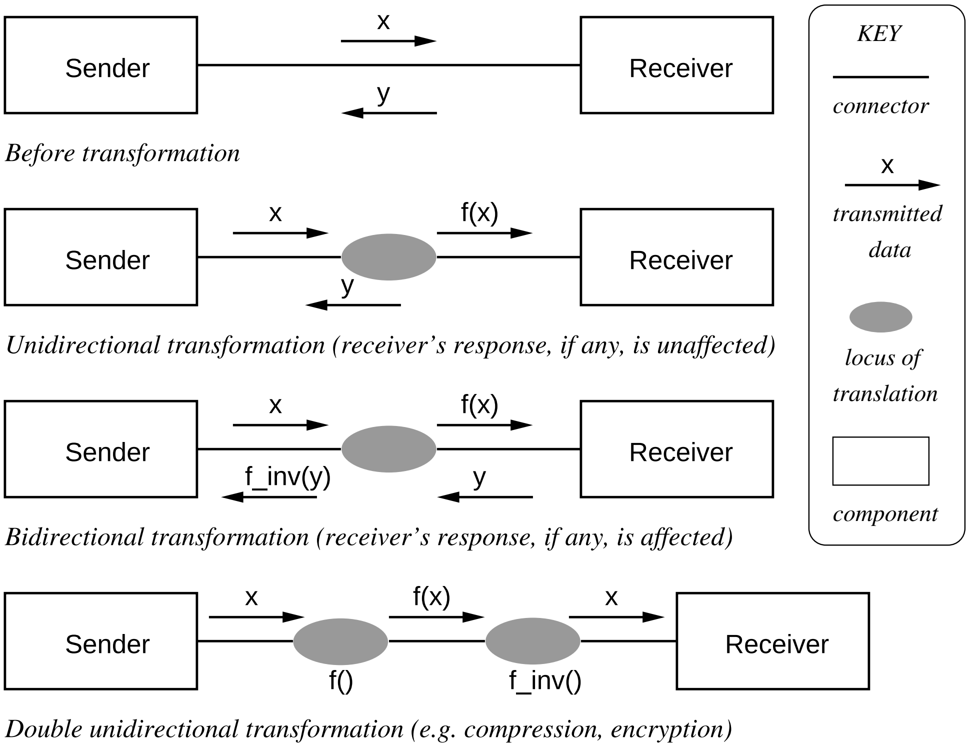

Figure 5.11: Sketch: Data translations in a two-role connector

5.4.1 Data Translation Template

The data translation template is a pattern for creating a data translation connector transformation[11]. The translation given in this template is associated with a specific role of the target connector. First the prerequisite information and the general form of the pattern will be described, then FSP descriptions and examples will be given.

A data translation may have one of two forms: it may be unidirectional or bidirectional. In a bidirectional data translation, the data sent from the role to the glue is translated according to some mapping f, and the data sent from the glue to the role is also translated according to some mapping f^{-1}, the inverse of f. In a unidirectional data translation, only one of the two directions (outgoing from the role or incoming to the role) is translated.

The bidirectional case requires a function[12] f (representing a domain-specific data

[11]Figures 3.2 through 3.6 are repeated here as Figures 5.11 through 5.30 for the reader's convenience.

[12]For translations that are more complex than a memoryless function, a *sessionize* transformation

translation) that maps events from alphabet A to an alphabet A', and an inverse function f^{-1}. A' may represent values that have been compressed, encrypted, converted to an intermediate representation (for translation between multiple mutually unintelligible formats), etc. For example, if the desired data translation is "compression", then f maps uncompressed data to compressed data, f^{-1} decompresses the data, and A' represents the range of compressed data; if the desired translation is "encryption", then f encrypts, f^{-1} decrypts, and A' represents the range of encrypted data. We may also have an additional constraint that, for any event e in A, $f(e) \neq e$ (e.g., to require that an encryption function encrypts all events), or a constraint that A and A' are disjoint. The unidirectional case requires one function which will be referred to here as f (if outgoing) or f^{-1} (if incoming); it need not have an inverse.

What will the structure of the transformation look like? For each role R experiencing the translation, we will add a wrapper process[13] W_R. W_R uses f to map initiated events from A (the role's alphabet) to A' and uses f' to map observed events from A' to A. Since FSP syntax does not enable us to distinguish between initiated and observed events, it will be necessary to identify these events explicitly. In the parameterized domain-specific transformation this is done by referring to *sets* of events which must be defined by the user of the parameterized transformation.

In addition, the transformation must also alter the glue's alphabet to match. Let E_I be the set of events initiated by role R, and E_O be the set of events observed by role R (these are the sets that would be defined in the parameterized domain-specific transformation). In the glue process G, any event e in E_I or E_O must be replaced by $f(e)$.

The effect of these two steps (the addition of the wrapper process W_R, and the remapping of a subset of the glue's alphabet) is to decouple R and G so that they no longer directly synchronize, but instead have their communication mediated by the translator W_R. Decoupling should be a familiar technique from earlier examples in this chapter.

An arbitrary mapping, e.g., $(\mathsf{a} \rightarrow \mathsf{m} \rightarrow W_R \mid \mathsf{b} \rightarrow \mathsf{n} \rightarrow W_R \mid \ldots)$, is somewhat cumbersome to explicitly state in FSP, though it would be possible to create a tool to automatically generate the process W_R from a more compact representation of the

may be a necessary complement to the data translation transformation, as seen with Kerberos authentication in section7.1.

[13]The notion of a wrapper process was introduced earlier in section 5.2.3.

```
// An example connector
Caller = (call → return → Caller).
Definer = (call → return → Definer).
Glue = (caller.call → definer.call → Glue
        | definer.return → caller.return → Glue).
||ProcCall = (caller:Caller || definer:Definer || Glue).
// Application of translation wrappers to example
set Role1 = {caller}
set Tag1 = {tag}
set CallerInitiates = {call}
set CallerObserves = {return}
TranslateCaller = ([Role1].[e:CallerInitiates] → [Tag1].[Role1].[e] → TranslateCaller
                  | [Tag1].[Role1].[e:CallerObserves] → [Role1].[e] → TranslateCaller).
set Role2 = {definer}
set Tag2 = {tag}
set DefinerInitiates = {return}
set DefinerObserves = {call}
TranslateDefiner = ([Role2].[e:DefinerInitiates] → [Tag2].[Role2].[e] → TranslateDefiner
                   | [Tag2].[Role2].[e:DefinerObserves] → [Role2].[e] → TranslateDefiner).
||TranslatedProcCall = (caller:Caller || definer:Definer ||
                        (Glue/{[Tag1].[r:Role1]/[r]})/{[Tag2].[r:Role2]/[r]}
                        || TranslateCaller || TranslateDefiner).
```

Figure 5.12: Data translation

function. When the precise function is not relevant to the analysis being performed, we may choose, when applying the data transformation template, to substitute a different function that can be stated more compactly in FSP. Let's consider this situation in more detail.

A common special case is the problem of checking which alphabet (A or A') is in use at a particular location in the connector protocol. For example, when composing a data transformation with another connector transformation, the order of composition affects which alphabet is "visible" to the other transformation; an inappropriate ordering could result in design errors such as inadvertent exposure of unencrypted data. In this case, we are not interested in the specific effects of the function except

// Data translation wrapper template
// This template is to be applied to some role R.
// It is applied by defining the sets R, E_I, E_O, and Tag
// (the sets may be renamed provided the name is changed throughout).
// R is a one-element set containing the role's prefix, e.g., caller.
// E_I is the set of the events initiated by R.
// E_O is the set of the events observed by R.
// Tag is a one-element set containing the tag, e.g., tag.
TranslateX $=$ ([R].[e:E_I] $\rightarrow$ [Tag].[R].[e] $\rightarrow$ TranslateX
| [Tag].[R].[e:E_O] $\rightarrow$ [R].[e] $\rightarrow$ TranslateX).
// General template for glue renaming
// The composite process will have this form:
$\|$*Connector* $= (r_1 : R_1 \parallel \ldots \parallel r_n : R_n \parallel$ TranslateX $\parallel \ldots$
$\parallel$ (Glue/{[Tag].[r:R].[e:E_I]/[r].[e]})/{[Tag].[r:R].[e:E_O]/[r].[e]}).
// A simpler template is possible
// if $E_I \cup E_O = \{$all events shared by role and glue$\}$:
$\|$*Connector* $= (r_1 : R_1 \parallel \ldots \parallel r_n : R_n \parallel$ TranslateX $\parallel \ldots \parallel$ Glue/{[Tag].[r:R]/[r]}).

Figure 5.13: Data translation template

insofar as it maps events in A to events in some new alphabet A'. We may represent this effect generically in FSP with a simple function f_t that prefixes a tag t to events; its inverse removes the tag t from tagged events. Such a mapping is illustrated in the example of Figure 5.12. The advantages of this simplification are that the A' produced by f_t is disjoint from A (making it trivial to determine which alphabet is in use) and that the mapping of each event need not be specified individually in W_R. The disadvantage is that any additional meaning that may have been present in the original mapping (e.g., an indication of what the purpose of the transformation may be) is lost.

Figure 5.12 depicts wrapper processes TranslateCaller and TranslateDefiner which illustrate the simplified tag-addition function f_t. TranslateCaller performs translation between the caller role and the glue, while TranslateDefiner mediates between the definer role and the glue. The two translation wrappers are nearly identical to one another, differing only in the sets with which they are parameterized. The glue has undergone one event-renaming for each of the two wrappers; here I have made an

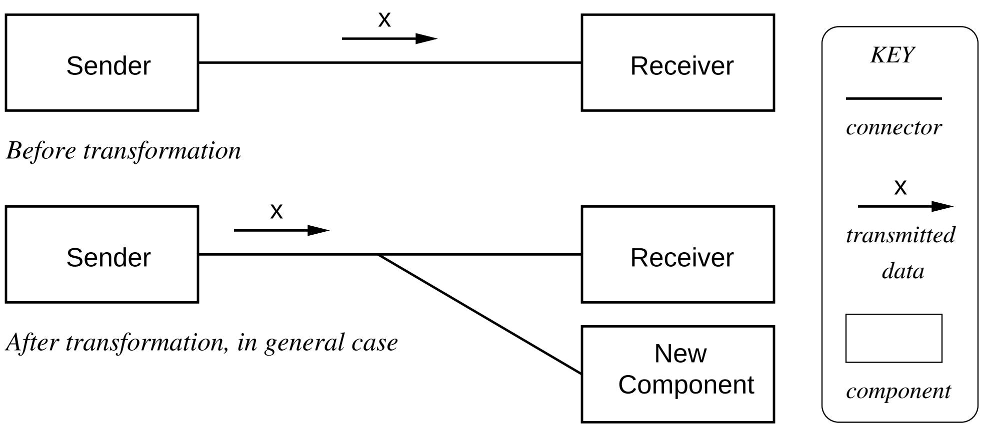

Figure 5.14: Sketch: Role addition in a two-role connector

assumption that $E_I \cup E_O$ (for either affected role) is equivalent to the set of events shared by the role and the glue. The more general renaming template, to be used if this assumption is not known to hold, is shown in Figure 5.13 and requires two renamings for each translation wrapper used.

As mentioned above, these particular templates are appropriate when one is primarily interested in determining whether an event *has* or *has not* been translated, rather than being interested in what the meaning or purpose of the translation is. Analyses that are appropriate for this transformation include checking which alphabet is visible at intermediate locations (e.g., at the glue); to give a domain-specific example, this analysis could reveal that encryption is being misused and unencrypted data can be seen at an intermediate location; or, it could indicate that this transformation and another transformation (such as Add Observer) have been composed in the wrong order, and a new role that expects to see uncompressed data is being subjected to compressed transmissions.

5.4.2 Role Addition Template

A role addition template adds a new role process plus a wrapper process to mediate between the new role and the existing glue; the wrapper process determines the degree of interaction available to the new role (does it listen only, intercept events, insert new events, etc.), preventing it from synchronizing directly with other processes, particularly other roles.

In the general case, we can characterize this transformation as adding one (or

```
// Observer pattern
// Observed is the set of events to monitor
set Observed = {role1.event, ...}
Observer = ([e:Observed] → Observer).
RelayObserver = ([e:Observed] → observer.[e] → RelayObserver).
||Connector = (role1:Role1 || ... || Glue || RelayObserver || observer:Observer).
```

Figure 5.15: Add Observer

more) new role processes ($R_{n+1} \ldots R_{n+k}$) and applying some function f_G to the existing glue process G. This expression may be familiar from section 5.2.3:

$$R_1 \| \ \ldots \| \ R_n \| \ f_G(G) \| \ R_{n+1} \| \ \ldots \| \ R_{n+k} \tag{5.9}$$

In the general case, little can be said about the *substructure* of these added or modified processes. However, in more specific cases, greater leverage can be gained; because their purpose is more constrained than in the general case, the added processes do have an identifiable substructure which can be reused to aid in the pattern-based writing of more domain-specific instantiations. I describe patterns for four kinds of role-addition transformations: Add Observer, Add Switch, Add Redirect, and Add Parallel.

Observer

The add-observer template adds a new role that *observes* events in a subset of the glue's alphabet. These events are relayed to the observer role by a wrapper process. Why use a wrapper process merely to dual-cast events when the new role could instead be allowed to participate[14] in the observed events directly? I chose to disallow this direct synchronization, both to provide better opportunities for checking and because allowing the observer to directly synchronize with other roles is not a good match for the implementation or for a connector designer's mental model of an unobtrusive "observer." Let's consider this issue in more detail. In the general case, one would expect the event ordering for the observer to be no more constrained than the event ordering for the observed role(s). Suppose that the person who writes the connector

[14] Recall that in FSP, concurrent processes that share events will synchronize on those events.

specification makes a mistake in specifying the observed role, so that the role accepts some inappropriate ordering of events. If the observer and the observed role are allowed to synchronize directly, and the observer does not accept this ordering of events, we have missed an opportunity to detect a problem, because the observer simply won't allow that ordering to occur. If they do not synchronize directly, and the observer receives an event relayed from the observed role that does not match the observer's notion of proper event ordering, a deadlock will result (which makes the mistake in the observed role difficult to miss). Furthermore, direct synchronization between the observer and another role does not in general have an equivalent in a connector *implementation* and is likely to be counterintuitive.

Figure 5.15 shows a template for a generic observer, Observer, and its wrapper RelayObserver. The Observer process shown here is very simple; it accepts events from the Observed set, in any order. One could instead restrict the observer to a specific event ordering, which would have much the same effect as writing a safety property: if events in the observer's alphabet occur in an order that is unacceptable to the observer, analysis will indicate a deadlock. (Depending on the use for which the observer component is intended and the severity of out-of-order events, it may be preferable for the component simply to log or report these violations while allowing the communication to continue, as the nonconstraining Observer role would do. For example, one might require that caller.call and caller.return events alternate, using the observer component to log any violations of this event ordering.)

Figure 5.16 shows an example connector to which an instantiation of a parameterized add-observer transformation has been applied. The user of the parameterized domain-specific transformation must define the set of events to be observed; we require Observed to be a subset of the glue's alphabet and thus all events should be qualified with the prefix of the existing role that shares the event with the glue. This enables the observer role to determine the origin of the event, if desired (if the prefix is not desired, a data translation wrapper may be added to strip the prefix).

If Observer does not impose an event ordering (and if Observed is a subset of the glue's alphabet, so that the new role's alphabet is disjoint from all of the original processes and the new wrapper shares events only with the glue and the new role), we expect that adding the Observer pattern to a connector will not affect the behavior of the connector with respect to the original roles. To check this expectation, we would take the new connector and the original connector and designate one as a safety

```
// Example connector
Caller = (call → return → Caller).
Definer = (call → return → Definer).
Glue = (caller.call → definer.call → Glue
        | definer.return → caller.return → Glue).
||ProcCall = (caller:Caller || definer:Definer || Glue).

// Applying Observer to example
set Observed = { caller.call, caller.return }
Observer = ([e:Observed] → Observer).
RelayObserver = ([e:Observed] → observer.[e] → RelayObserver).
||ProcCall = (caller:Caller || definer:Definer || Glue
              || RelayObserver || observer:Observer).
```

Figure 5.16: Applying Add Observer

property of the other (ascertaining, for example, whether the two connectors might allow events to occur in a different order).

Switch

In an add-switch transformation, the added role R_n is similar in interface to an existing role R_i (or perhaps multiple existing roles). One of the set of similar roles is "active" at any particular time. The transformation adds a process to direct traffic to the active role or, when a triggering event is received, to switch which role is active.

Figure 5.17 shows a simplified template followed by examples of use. This template does not include storing an event and replaying it after the trigger (as seen in the earlier Failover examples); instead the event that (re)initializes communication on the switched role is drawn from a set (Start1 or Start2).

In Figure 5.17, the composite process SwitchingConn1 shows the switching wrapper interposed between the glue and the set of similar roles, which enables the set of roles to appear to the glue to be one role. Events pass from other roles to the glue to the switching wrapper to one of the set of similar roles (and the reverse path). SwitchingConn2 is an alternative arrangement in which events pass from other roles to the switching wrapper, then to one of a set of glues, then to one of the set of

```
// Simplified template
Switch = Case1,
        Case1 = ([r:R].[e:RInitiates] → case1.[r].[e] → Case1
                | case1.[r:R].[e:RObserves] → [r].[e] → Case1
                | case1.[r:R].[e:Switch1] → case2.[r].[Start2] → Case2),
        Case2 = ([r:R].[e:RInitiates] → case2.[r].[e] → Case2
                | case2.[r:R].[e:RObserves] → [r].[e] → Case2
                | case2.[r:R].[e:Switch2] → case1.[r].[Start1] → Case1).

// Placed between the glue and the set of switching roles:
// (see Figure 5.18, top)
set R = {definer}
||SwitchingConn1 = ( ...case1.definer:Definer || case2.definer:Definer
                 || Glue || Switch).

// Placed between the other roles and the glue:
// (see Figure 5.18, bottom)
set R = {caller}
||SwitchingConn2 = ( ...case1.definer:Definer || case2.definer:Definer
                 || case1:Glue || case2:Glue || Switch).
```

Figure 5.17: Add Switch

similar roles: in effect we have pushed the glue process to the other side of the switching wrapper. A desired ordering of connector transformations may dictate the use of this alternative arrangement. For example, in the combination of Retry and Failover culminating in Figure 5.6, Retry must be used between the client role and the glue to hide the glue's transient errors and must have access to errors before they become available to Failover; thus Failover must use a switching wrapper on the "far side" of the glue from the servers.

In Figure 5.19 there is a generalized template that can store and replay an event; this is suitable for the case when the incentive to switch comes from the active role (perhaps while it is providing service) and the event that triggered the service needs to be repeated to get the next active role into the right state.

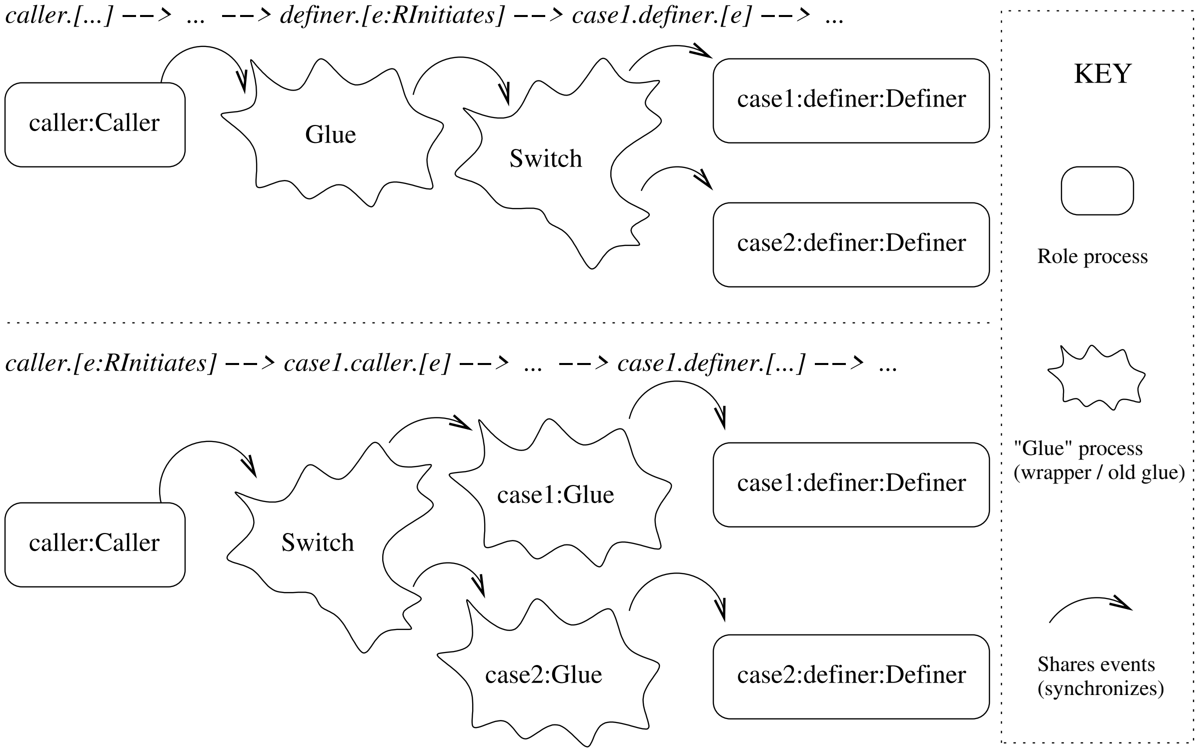

Figure 5.18: Clarification of Add Switch wrapper placement

// General case - see also Sessionize
Switch = (hide.[r:R].[e:RInitiates] $\rightarrow$ Case1[r][e]),
  Case1[rsave:R][save:RInitiates] =
    ([r:R].[e:RInitiates] $\rightarrow$ case1.[r].[e] $\rightarrow$ Case1[r][e]
    | case1.[r:R].[e:RObserves] $\rightarrow$ [r].[e] $\rightarrow$ Case1[rsave][save]
    | case1.[r:R].[e:Switch1] $\rightarrow$ case2.[rsave].[save] $\rightarrow$ Case2[rsave][save]),
  Case2[rsave:R][save:RInitiates] =
    ([r:R].[e:RInitiates] $\rightarrow$ case2.[r].[e] $\rightarrow$ Case2[r][e]
    | case2.[r:R].[e:RObserves] $\rightarrow$ [r].[e] $\rightarrow$ Case2[rsave][save]
    | case2.[r:R].[e:Switch2] $\rightarrow$ case1.[rsave].[save] $\rightarrow$ Case1[rsave][save])\{hide}.

Figure 5.19: Add Switch, general case

Note that the aggregation transformation bears some similarity to the arrangement in SwitchingConn2; it, too, combines a set of glues so that one glue is in active use. This resemblance suggests that, for a given task which is to be achieved by the

```
// The redirecter is associated with a specific existing role R.
// The RedirectW wrapper mediates between R and the glue.
// As seen in previous templates,
// R is a one-element set containing the role's prefix, e.g., caller.
// E_I is the set of the events initiated by R.
// E_O is the set of the events observed by R.
// Tag is a one-element set containing the tag, e.g., tag.
RedirectW = ([r:R].[e:E_I] → chooser.[e] → RedirectW
            | chooser.back.[e:E_O] → [r:R].[e] → RedirectW
            | chooser.fwd.[e:E_I] → [Tag].[r:R].[e] → RedirectW
            | [Tag].[r:R].[e:E_O] → chooser.observe.[e] → [r][e] → RedirectW).
Chooser = ([e:E_I] → ( hide → back.[e2:E_O] → Chooser
                     | hide → fwd.[e] → Chooser )
          | observe.[e:E_O] → Chooser)\{hide}.
// R's events in the glue must be tagged with tag.
// The composite process will have this form:
||Connector = (r_1 : R_1 || ... || r_n : R_n
             || chooser:Chooser || RedirectW
             || (Glue/{[Tag].[r:R].[e:E_I]/[r].[e]})/{[Tag].[r:R].[e:E_O]/[r].[e]} ).
```

Figure 5.20: Add Redirect

application of connector transformations, the collection of transformations to be used is not necessarily uniquely dictated.

This transformation assumes that the similar role processes have (ultimately) disjoint alphabets; this is enforced by the addition of prefixes, e.g., the case1 and case2 prefixes shown in 5.17. One might also wish to compare the behavior of the original role to the behavior of the Switch plus set of similar roles. This comparison may be assisted by using the original role as a basis for a safety property.

Redirect

An add-redirect transformation introduces a new role to which a subset of intercepted communication traffic is *redirected* (e.g., to confirm or authorize a request), after which the intercepted event may be sent on to its original destination. The transformation

$$
\begin{aligned}
\mathsf{CacheW} = (&[\mathsf{r}{:}R].[\mathsf{e}{:}E_I] \rightarrow \mathsf{cache}.[\mathsf{e}] \rightarrow \mathsf{CacheW}\\
&\mid \mathsf{cache.hit}.[\mathsf{e}{:}E_O] \rightarrow [\mathsf{r}{:}R].[\mathsf{e}] \rightarrow \mathsf{CacheW}\\
&\mid \mathsf{cache.miss}.[\mathsf{e}{:}E_I] \rightarrow [\mathsf{Tag}].[\mathsf{r}{:}R].[\mathsf{e}] \rightarrow \mathsf{CacheW}\\
&\mid [\mathsf{Tag}].[\mathsf{r}{:}R].[\mathsf{e}{:}E_O] \rightarrow \mathsf{cache.add}.[\mathsf{e}] \rightarrow [\mathsf{r}][\mathsf{e}] \rightarrow \mathsf{CacheW}).
\end{aligned}
$$

// Assumes one request outstanding; component can match it to response

$$
\begin{aligned}
\mathsf{Cache} = (&[\mathsf{e}{:}E_I] \rightarrow (\ \mathsf{hide} \rightarrow \mathsf{hit}.[\mathsf{e2}{:}E_O] \rightarrow \mathsf{Cache}\\
&\qquad\qquad \mid \mathsf{hide} \rightarrow \mathsf{miss}.[\mathsf{e}] \rightarrow \mathsf{Cache}\ )\\
&\mid \mathsf{add}.[\mathsf{e}{:}E_O] \rightarrow \mathsf{Cache})\backslash\{\mathsf{hide}\}.
\end{aligned}
$$

Figure 5.21: Add Redirect: redirect to Cache

also adds a wrapper process which mediates communication between this role and the processes of the original connector. A redirection template is shown in Figure 5.20; this redirection is associated with one existing role R of the connector and redirects its outgoing traffic to the new role. In this figure, the new role is represented by the Chooser process.

Leaving aside the form of the new role for the moment, let us consider the wrapper process. Like other wrappers we have seen so far, it decouples existing processes (via event relabelling) and mediates their communication. If an event initiated by the affected existing role is not in the set of events to divert, then the event will simply be relayed to the original destination; otherwise, it will be redirected to the new role. (This discrimination between relayed and redirected events may appear explicitly in the wrapper process, or, as in the template given in Figure 5.20, the wrapper may divert all events and leave the task of differentiation to the added component.) The wrapper process also has the task of relaying the new role's responses to the sender or to the original destination of the redirected event. Finally, the wrapper process allows the new role to observe the affected role's incoming events (whether these events are of interest to the new component depends on the domain-specific use of the transformation).

The new role engages in events in the set of redirected events and initiates events in response, either sending an event "forward" to the glue, or "back" to the role that sent the redirected event. The nature of the choice between forward and back is determined by the domain-specific use of the transformation. For example, the new role may enable user-level confirmation of dangerous actions or program-level authorization of restricted services; an event is relayed forward if it has been authorized so that the request may proceed, or, if authorization is denied, a different event $e2$ (perhaps indicating the error) is sent back to the originator of the request. A second example appears in Figure 5.21 (essentially identical to the template of Figure 5.20, with some event labels changed for clarity); here the new role Cache represents a kind of local cache, which can send back cached results or send forward a request for which no suitable cached result is available.

Analyses suitable for this transformation include comparing the new behavior to the original behavior (for example, if the new role is constrained to relay all events, i.e., back events are disallowed, then its behavior should conform to that of the original connector) as well as checking domain-specific assertions such as "events in the set X are always subjected to authorization before reaching role R" (which may be phrased as an event ordering in a safety property).

Parallel replication

Like the add-switch transformation, this transformation adds a role R_n that is similar in interface to an existing role R_i. This transformation differs by not designating a single "active" role; rather, traffic directed to R_i is broadcast to *each* role in the set of similar roles. The transformation is responsible for introducing this multiplexing and, if a response is generated, for converting multiple responses into a single response. Figure 5.22 illustrates a pattern in which a Split wrapper broadcasts to the set of similar roles, and a Join wrapper collects the responses. In this example, Join requires all responses to be collected before relaying a single response to the original glue, and the policy for determining the value of the collated response is embodied in the local process Finish (in this case it simply returns the last response received).

5.4.3 Aggregate Template

The aggregate template combines two connectors into one. The original connectors must have corresponding roles. The transformation adds a new process whose task is to decide which glue to use; one glue may be in use at a time. A decision is made in an initial negotiation and may be revisited at specified points in the communication. (Note that the user of the formalism must specify these points of transition; I do not automatically determine them.)

```
// Consider a simple generic client-server connector.
// The server S1 selects a return value from the set Values.
set Values = {...}
C1 = (call → return.[v:Values] → C1).
S1 = (call → select.[v:Values] → return.[v] → S1)\{select}.
Glue = (client.call → server.call → Glue
         | server.return.[v:Values] → client.return.[v] → Glue).
||C1S1 = (client:C1 || server:S1 || Glue).

// Replicate the server using Split and Join wrappers.
set ServerObserves = {call}
Split = (tag.server.[e:ServerObserves] → s1.[e] → s2.[e] → s3.[e] → Split).
Join = (s1.return.[x:Values] → Jgot1[x]
        | s2.return.[x:Values] → Jgot2[x]
        | s3.return.[x:Values] → Jgot3[x]),
    Jgot1[x:Values] = (s2.return.[y:Values] → Jneed3[x][y]
                      | s3.return.[y:Values] → Jneed2[x][y]),
    Jgot2[x:Values] = (s1.return.[y:Values] → Jneed3[x][y]
                      | s3.return.[y:Values] → Jneed1[x][y]),
    Jgot3[x:Values] = (s1.return.[y:Values] → Jneed2[x][y]
                      | s2.return.[y:Values] → Jneed1[x][y]),
    Jneed1[x:Values][y:Values] = (s1.return.[z:Values] → Finish[x][y][z]),
    Jneed2[x:Values][y:Values] = (s2.return.[z:Values] → Finish[x][y][z]),
    Jneed3[x:Values][y:Values] = (s3.return.[z:Values] → Finish[x][y][z]),
    Finish[x:Values][y:Values][z:Values] = (tag.return.[z] → Join).
||C1S3 = (client:C1 || s1:S1 || s2:S1 || s3:S1 || Split || Join || Glue/{tag.server/server}).
```

Figure 5.22: Add Parallel: Voting

There are several pieces of this transformation to consider: how the roles of the transformed connector are produced, how the glues are combined, and how the negotiation may be performed.

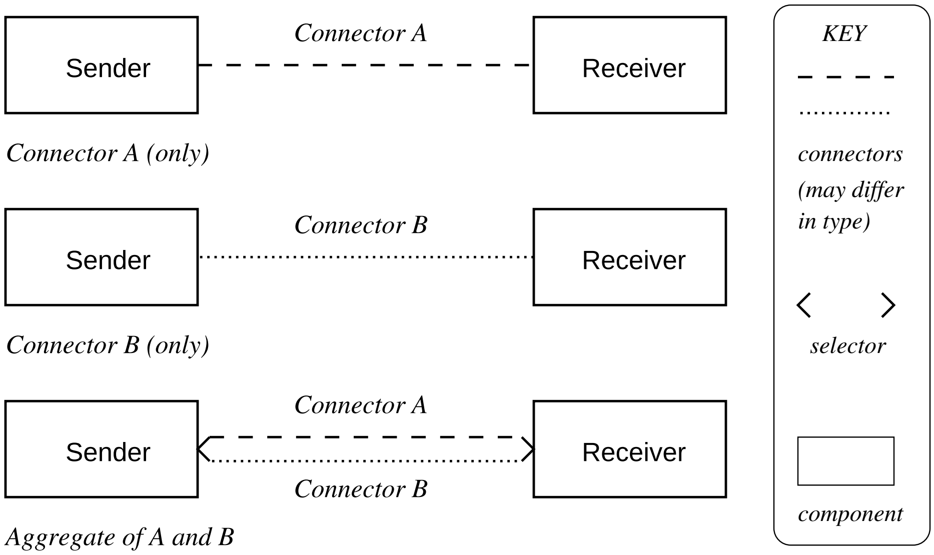

Figure 5.23: Sketch: Aggregate of two two-role connectors

Aggregation of roles

The aggregate transformation requires the two original connectors to have a one-to-one correspondence between roles. The roles of the resulting connector may have one of two forms. In the first case, corresponding roles in the original connector may be *identical* to one another; in this case there is no need to combine them. In the second case, each role in a pair of non-identical corresponding roles becomes a local process in the definition of the transformed role, which "switches" between the two original behaviors (a pattern of behavior which may be familiar from the wrapper of Figure 5.17).

Figure 5.24 assumes the first case, in which the corresponding roles are identical. Figure 5.25 illustrates a pattern for role-aggregation in the second case; local processes R1A and R1B can only be used unchanged if the connector negotiates only at the beginning, and otherwise they must be altered to add a [trigger] choice element (similar to the wrappers).

Aggregation of glues

If we continue the line of approach taken in the templates seen so far, the techniques to use for aggregating glues may already be fairly evident. First, we require that the glues should be unable to directly synchronize with one another, since only one glue

// $E_n A$ should be the set of events shared by role n and glue A. set $E_1 A = \{\dots\}$
// Tag$A$ is a unique tag used to label glue A
set $\mathsf{TagA} = \{\mathit{tag1}\}$
set $\mathsf{TagB} = \{\mathit{tag2}\}$
// The transformation requires a wrapper process for each role.
// W_1 is a template for R_1's wrapper.
// The wrapper contains a local process for each glue.
$W_1 = (\ \mathit{glueA} \rightarrow \mathsf{RelayToGlueA} \mid \mathit{glueB} \rightarrow \mathsf{RelayToGlueB}\ ),$
$\quad \mathsf{ToGlueA} = (\ [\mathsf{e}{:}E_1 A] \rightarrow [\mathsf{TagA}].[\mathsf{e}] \rightarrow \mathsf{ToGlueA} \mid [\mathsf{Trigger}] \rightarrow W_1),$
$\quad \mathsf{ToGlueB} = (\ [\mathsf{e}{:}E_1 B] \rightarrow [\mathsf{TagB}].[\mathsf{e}] \rightarrow \mathsf{ToGlueB} \mid [\mathsf{Trigger}] \rightarrow W_1).$
$W_2 = (\dots).$
$\dots$
// The transformation also requires a decision-maker process.
// *Select* represents the least-constrained possible decision-maker.
// It nondeterministically selects the event that (re)initializes the wrappers $W_1 \dots W_n$.
$\mathsf{Select} = (\mathsf{hide} \rightarrow \mathit{glueA} \rightarrow \mathsf{Select} \mid \mathsf{hide} \rightarrow \mathit{glueB} \rightarrow \mathsf{Select}) \backslash \{\mathsf{hide}\}.$
// The composite process will have this form:
$\|\mathit{Connector} = (r_1 : R_1 \parallel W_1 \parallel \dots \parallel r_n : R_n \parallel W_n$
$\qquad \dots \parallel \mathsf{Select} \parallel [\mathsf{TagA}]{:}\mathsf{GlueA} \parallel [\mathsf{TagB}]{:}\mathsf{GlueB}).$

Figure 5.24: Aggregate

is to be "in use" at a time. To ensure that their alphabets are disjoint, therefore, we will add a tag to each of the glue processes. Second, adding a tag also prevents the glue processes from synchronizing with the roles; we will add wrapper processes to mediate between the roles and the tagged glue processes. Each role will have an associated wrapper which directs its events to the "active" glue.

Switching

Two things must still be clarified. Who decides which glue is active? How does a wrapper process know which glue is supposed to be active? The source of the decision, and the inputs to the decision, depend on the domain-specific use of this transformation; in the template shown in Figure 5.24, the decision maker is represented by an additional process. When a decision is made, a notification event is sent to each of

// Given original roles R_{1A} and R_{1B}:
// R_{1A} is a role from connector A,
// R_{1B} is the corresponding role from connector B.
// (For illustration, a possible definition of each role is given in italics.)
$\mathsf{R1A} = (\ \textit{request} \rightarrow \textit{response} \rightarrow \mathsf{R1A}\ ).$
$\mathsf{R1B} = (\ \textit{query} \rightarrow (\textit{answer} \rightarrow \mathsf{R1B} \mid \textit{error} \rightarrow \mathsf{R1B})\ ).$

// To produce the aggregated role R_1 in the aggregated connector,
// construct a new process in which R1A and R1B
// are local processes.
$\mathsf{R1} = (\mathsf{glueA} \rightarrow \mathsf{R1A}$
$\quad\mid \mathsf{glueB} \rightarrow \mathsf{R1B}\ ),$
$\quad\mathsf{R1A} = (\ \textit{request} \rightarrow \textit{response} \rightarrow \mathsf{R1A}\ ),$
$\quad\mathsf{R1B} = (\ \textit{query} \rightarrow (\textit{answer} \rightarrow \mathsf{R1B} \mid \textit{error} \rightarrow \mathsf{R1B})\ ).$

Figure 5.25: Role Aggregation

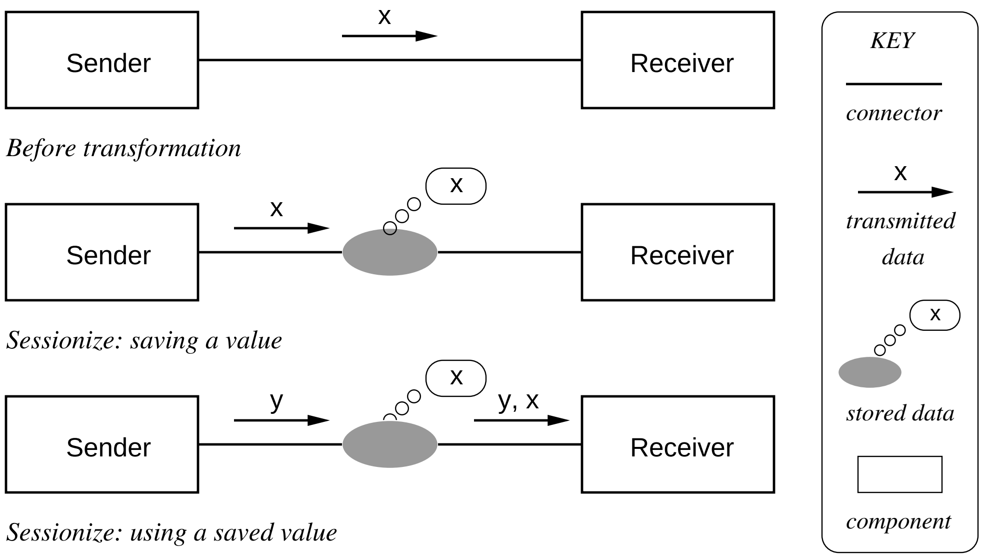

Figure 5.26: Sketch: Sessionize, saving and replaying

the wrappers.

$$
\begin{aligned}
&\mathsf{Session} = ([\mathsf{e{:}NormalEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{e}] \rightarrow \mathsf{Session} \\
&\qquad\qquad \mid [\mathsf{s{:}SavedEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{s}] \rightarrow \mathsf{InSession}[\mathsf{s}]), \\
&\mathsf{InSession}[\mathsf{saved{:}SavedEvents}] = ([\mathsf{e{:}NormalEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{e}] \rightarrow \mathsf{InSession}[\mathsf{saved}] \\
&\qquad\qquad \mid [\mathsf{s{:}SavedEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{s}] \rightarrow \mathsf{InSession}[\mathsf{s}]).
\end{aligned}
$$

Figure 5.27: Pattern: Saving a Value

$$
\begin{aligned}
&\mathsf{Session} = ([\mathsf{e{:}NormalEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{e}] \rightarrow \mathsf{Session} \\
&\qquad\qquad \mid [\mathsf{s{:}SavedEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{s}] \rightarrow \mathsf{InSession}[\mathsf{s}]), \\
&\mathsf{InSession}[\mathsf{saved{:}SavedEvents}] = ([\mathsf{e{:}NormalEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{e}].[\mathsf{saved}] \rightarrow \mathsf{InSession}[\mathsf{saved}] \\
&\qquad\qquad \mid [\mathsf{s{:}SavedEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{s}] \rightarrow \mathsf{InSession}[\mathsf{s}]).
\end{aligned}
$$

Figure 5.28: Pattern: Continual Replay

5.4.4 Sessionize

The sessionize template creates additional state in a connector protocol. A piece of information is recorded at the beginning of a "session" and may be replayed in subsequent transmissions. In FSP the information is taken from an event and is recorded in a parameter which is added by the transformation; it may be replayed by using the parameter.

Figure 5.27 illustrates the basic pattern for *saving* an event within a session. (The saved value is not used.) A wrapper such as this would be placed between a role and the glue to mediate between them.[15] Events in the NormalEvents set are relayed without affecting the saved value. Events in the SavedEvents set are used to update the saved value; in the given example, these events are also relayed.

Figure 5.28 shows one possible form of use of a saved value. In this pattern, the saved value is replayed with every event communicated inside the session. Here it is shown as a rider on the event, but could also be sent separately either before or after the event. A domain-specific application of this replaying pattern is authentication, where the saved value constitutes a proof of identity (which may be used in composition with a data translation transformation to add a digital signature to events).

[15]For brevity, this figure considers only the role-to-glue directed events; the reader should be familiar by now with the construction of wrappers that mediate glue-to-role events as well.

$$
\begin{aligned}
&\mathsf{Session} = ([\mathsf{e{:}NormalEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{e}] \rightarrow \mathsf{Session}\\
&\qquad\qquad \mid [\mathsf{s{:}SavedEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{s}] \rightarrow \mathsf{InSession}[\mathsf{s}]),\\
&\mathsf{InSession}[\mathsf{saved{:}SavedEvents}] = ([\mathsf{e{:}NormalEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{e}] \rightarrow \mathsf{InSession}[\mathsf{saved}]\\
&\qquad\qquad \mid [\mathsf{t{:}TriggerReplay}] \rightarrow [\mathsf{Tag}].[\mathsf{saved}] \rightarrow \mathsf{InSession}[\mathsf{saved}]).\\
&\qquad\qquad \mid [\mathsf{s{:}SavedEvents}] \rightarrow [\mathsf{Tag}].[\mathsf{s}] \rightarrow \mathsf{InSession}[\mathsf{s}]).
\end{aligned}
$$

Figure 5.29: Pattern: Conditional Replay

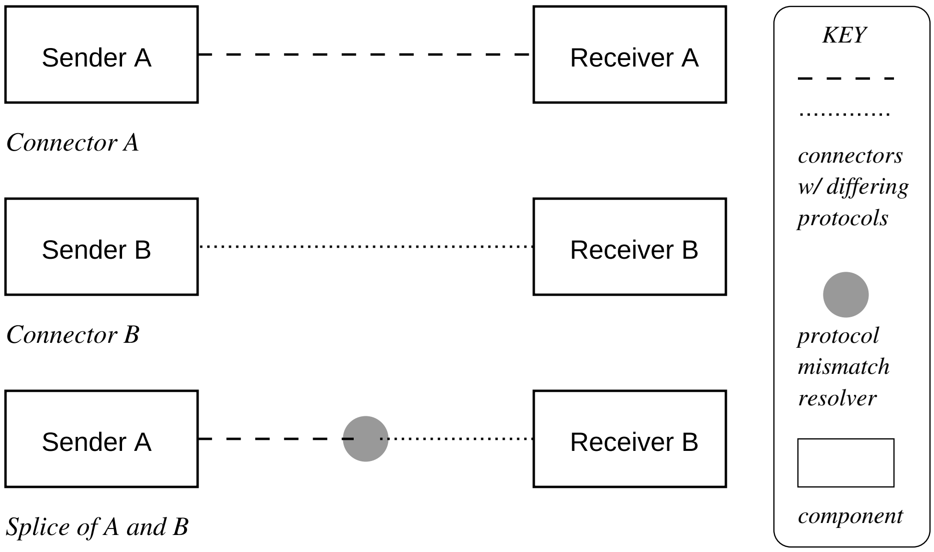

Figure 5.30: Sketch: Splice of two two-role connectors

Figure 5.29 shows another possible use of a saved value. In this pattern, the saved value is replayed only conditionally, and its replay is triggered by an event in the TriggerReplay set. (Here the transmission of the saved value *replaces* the trigger event, which is not relayed; this need not be the case.) This is similar to the event replay seen earlier in the Retry example, in which "error" events are replaced by saved values, with the effect of "retrying" a failed request.

5.4.5 Splice

The splice transformation combines connectors whose role protocols differ, thus enabling communication between some kinds of otherwise incompatible components. For example, given a component A that uses the SimpleSource interface in Figure 5.31 and a component B that uses the Sink interface, neither Pipe1 nor Pipe2 can be used

```
// Simple pipe: no open or close
SimpleSource = (write → SimpleSource).
SimpleSink = (read → SimpleSink).
SimpleGlue = (source.write → sink.read → SimpleGlue).
||Pipe1 = (source:SimpleSource || sink:SimpleSink || SimpleGlue).

Source = (open → OpenSource),
    OpenSource = (write → OpenSource | close → Source).
Sink = (open → OpenSink),
    OpenSink = (write → OpenSink | close → Sink).

// Push or pull is determined by event ordering in the glue:
Glue = ( source.write → sink.read → Glue
       | source.open → sink.open → Glue
       | source.close → sink.close → Glue ).
||Pipe2 = (source:Source || sink:Sink || Glue).
```

Figure 5.31: Two Pipes

to connect A and B. However, we might hope to *splice* Pipe1 and Pipe2 to produce a new connector Pipe12 whose roles are SimpleSource and Sink. The glue of Pipe12 must somehow overcome the disparity between its two ports.

In the general case it may not be straightforward or even possible to combine two arbitrary connectors. However, some specific cases are worth noting here: first, the expanding/compacting of a sequence of events, and second, the insertion/removal of an event that is not shared between the two connector protocols.

Expanding or compacting

Expanding one event into a sequence of events (or compacting an ordered sequence of events into one event) can be accomplished with the use of a wrapper process and one of the two glues. Consider the example in Figure 5.32. The Pipe3 connector example conveys sensor readings from three sensors in turn, in contrast to the Pipe3in1 connector which conveys all three readings in a single event. This situation is simplified by the assumption that Pipe3 always sends the values in the same order and reads

```
set Data = {a, ...}
Source3 = (sensor1.[Data] → sensor2.[Data] → sensor3.[Data] → Source3).
Sink3 = (sensor1.[Data] → sensor2.[Data] → sensor3.[Data] → Sink3).
Glue3 = (source.[s:Sensor].[d:Data] → sink.[s].[d] → Glue3).
||Pipe3 = (source:Source3 || sink:Sink3 || Glue3).

Source3in1 = (sensor.[Data].[Data].[Data] → Source3in1).
Sink3in1 = (sensor.[Data].[Data].[Data] → Sink3in1).
Glue3in1 = (source.sensor.[d1:Data].[d2:Data].[d3:Data]
           → sink.sensor.[d1].[d2].[d3] → Glue3in1).
||Pipe3in1 = (source:Source3in1 || sink:Sink3in1 || Glue3in1).

// Expanding the “3in1” event:
set Role = {source}
set Tag = {tag}
Expand = ([Role].sensor.[d1:Data].[d2:Data].[d3:Data] → [Tag].[Role].sensor1.[d1]
         → [Tag].[Role].sensor2.[d2] → [Tag].[Role].sensor3.[d3] → Expand).
||Expanded = (source:Source3in1 || sink:Sink3 || Glue3/{[Tag].[Role]/[Role]} || Expand).

// Compacting the “3” events:
Compact = ([Role].sensor1.[d1:Data] → [Role].sensor2.[d2:Data] → [Role].sensor3.[d3:Data]
          → [Tag].[Role].sensor.[d1].[d2].[d3] → Compact).
||Compacted = (source:Source3 || sink:Sink3in1 || Glue3in1/{[Tag].[Role]/[Role]} || Compact).
```

Figure 5.32: Expanding/compacting ordered event sequences

each sensor with equal frequency.[16]

A wrapper such as Expand or Compact can be used to splice Pipe3 and Pipe3in1. The Expand wrapper mediates between a role that has “fewer” events and a glue that has “more” events. Here is a more generalized expression for the pattern of which the Expand wrapper is an example:

[16]If this is not the case, the wrapper must cache the most recent value from each sensor. In addition, the wrapper must necessarily embody an update policy such as “send only when all have been updated” or “send when any have been updated.”

P = ([s:SharedEvents] $\rightarrow$ [Tag].[s] $\rightarrow$ P | $X \rightarrow Y \rightarrow$P).

Here SharedEvents is a set of events that are common between the two connectors and do not need to be modified, X represents an event e that is to be expanded, and Y represents the event sequence [Tag].e_1, [Tag].e_2, ...[Tag].e_n into which e is to be expanded.

P = ([s:SharedEvents] $\rightarrow$ [Tag].[s] $\rightarrow$ P | $e \rightarrow$ [Tag].e_1 ... $\rightarrow$ P).

One might prefer to express the expansion part of the wrapper as a sequence of successfully terminating processes $X; Y; P$ (where $X = (e\rightarrow$ END), etc). Unfortunately this is not convenient when the content of an event in $e_1 \dots e_n$ relies on information drawn from e, as is the case in the example seen in Figure 5.32.

The Compact wrapper is similar; X and Y are reversed so that a finite sequence of (untagged) events Y is compacted into a single (tagged) event X. This wrapper mediates between a role that has "more" events and a glue that has "fewer" events.

Suitable analyses associated with this transformation include confirmation of role compatibility in the resulting connector. Since the goal of the transformation is to produce a connector whose roles each match one of the roles of the original connectors, it may be desirable to check that this really is the case. This can be done by using these roles as safety properties of the resulting connector.

Inserting or removing

As seen above, a mismatch between an event e_s in one connector and a corresponding *event sequence* in another connector can be overcome with a wrapper. Inserting or removing an event e, where e appears in one connector but has *no equivalent* in the other connector, may seem an equally innocuous operation but actually presents significant potential problems.

Figure 5.31 is an example of this case. The Pipe2 connector's roles and glue have two events, open and close, which have no equivalent in the Pipe1 connector (Pipe1 assumes that reads and writes occur without need for opening and closing the pipe).

Given a connector A with roles R_{1A}, R_{2A} and glue G_A, and a connector B with roles R_{1B}, R_{2B} and glue G_B, where we wish to create a spliced connector that presents roles R_{1A}, R_{2B}, one might bridge the connectors by providing a bridge process B,

// A glue-bridging template for splicing connectors X and Y.
// Events that are initiated on the "left" side of the splice:
set LeftToRight = $\{\ldots\}$
// Events that are initiated on the "right" side of the splice:
set RightToLeft = $\{\ldots\}$
// Single-element sets containing a unique tag:
set LeftTag = $\{\ldots\}$
set RightTag = $\{\ldots\}$
Bridge = ([LeftTag].[e:LeftToRight] $\rightarrow$ [RightTag].[e] $\rightarrow$ Bridge
    | [RightTag].[e:RightToLeft] $\rightarrow$ [LeftTag].[e] $\rightarrow$ Bridge).
||RoleXToRoleY = ([LeftTag]:(*roleX:RoleX* || *GlueX*) || Bridge
    || [RightTag]:(*roleY:RoleY* || *GlueY*)).

Figure 5.33: Splice: glue-bridging template

either between the two glues G_A, G_B (Figure 5.33) or between the intermediate roles R_{2A}, R_{1B}. Both alternatives present difficulties.

The first alternative, glue-bridging, may prove inadequate when one of the glues relies on the presence of its intermediate role to provide event ordering constraints. The event ordering of a protocol (e.g., every call event must be followed by a return event and not by an init event) is not necessarily explicit in the glue; the original writer of the connector protocol may have decided to make the glue more permissive than the roles.[17] If we remove one of the roles, then relying upon the glue and the remaining role is not guaranteed to provide sufficient constraint, particularly when dealing with an event that did not appear in the second connector's alphabet and therefore is not constrained by the second connector's glue or role. Deadlock may result; another possible outcome is undesirable event orderings, which can be detected with safety properties. Figure 5.34 shows a pair of connectors that exhibit deadlock when spliced with glue-bridging.

The second alternative, role-bridging, provides greater constraint on a liberal glue. (Figure 5.35 shows the addition of intermediate roles into the underconstrained Role1AToRole2B spliced connector of Figure 5.34; adding Role2A constrains the init event sufficiently to prevent deadlock.) However, role-bridging can produce undesir-

[17]For a brief discussion of glue permissiveness, see section 5.2.2.

```
// Connector 1: role A queries, role B responds with a nondeterministic choice.
Role1A = (query → (yes → Role1A | no → Role1A)).
Role1B = (query → (hide → yes → Role1B | hide → no → Role1B))\{hide}.
Glue1 = (a.query → b.query → Glue1 | b.yes → a.yes → Glue1 | b.no → a.no → Glue1).
||C1 = (a:Role1A || b:Role1B || Glue1).
// Connector 2: similar, with (re)initialization.
Role2A = (query → (yes → Role2A | no → Role2A)
          | init → Role2A).
Role2B = (query → (hide → yes → Role2B | hide → no → Role2B)
          | init → Role2B)\{hide}.
Glue2 = (a.query → b.query → Glue2 | b.yes → a.yes → Glue2 | b.no → a.no → Glue2
         | a.init → b.init → Glue2).
||C2 = (a:Role2A || b:Role2B || Glue2).
// We can bridge from Role2A to Role1B with this template:
set LeftToRight = {a.query, b.query}
set RightToLeft = {b.yes, a.yes, b.no, a.no}
set LeftTag = {tag2}
set RightTag = {tag1}
Bridge = ([LeftTag].[e:LeftToRight] → [RightTag].[e] → Bridge
         | [RightTag].[e:RightToLeft] → [LeftTag].[e] → Bridge).
||Role2AToRole1B = ( [LeftTag]:(a:Role2A || Glue2) || Bridge || [RightTag]:(b:Role1B || Glue1) ).
// We cannot use the same template to bridge from Role1A to Role2B, because
// nothing prevents Glue2 from inserting init after query and deadlocking:
Bridge = ([LeftTag].[e:LeftToRight] → [RightTag].[e] → Bridge
         | [RightTag].[e:RightToLeft] → [LeftTag].[e] → Bridge).
||Role1AToRole2B = ( [LeftTag]:(a:Role1A || Glue1) || Bridge || [RightTag]:(b:Role2B || Glue2) ).
```

Figure 5.34: Underconstraint produces deadlock

able results when one of the intermediate roles is nondeterministic. This intermediate role (e.g., R_{2A}) originally represented an interface bound to a component that is making some choices, but now there is another role (e.g., R_{2B}) that stands in for the component; if both role processes are present and allowed to make nondeterministic choices, those choices may conflict. That is, the inclusion of the intermediate roles may result in an overconstrained composite process that will deadlock. For example,

```
// In this case, Role2A constrains init enough to prevent deadlock.
||Role1AToRole2Badd1 = ( [LeftTag]:(a:Role1A || Glue1) || Bridge
                         || [RightTag]:(b:Role2B || Glue2 || a:Role2A) ).

// If we also needed Role1B, however, a new deadlock would result
// from conflict between nondeterministic processes.
||Role1AToRole2Badd2 = ( [LeftTag]:(a:Role1A || Glue1 || b:Role1B) || Bridge
                         || [RightTag]:(b:Role2B || Glue2 || a:Role2A) ).
```

Figure 5.35: Intermediate roles provide more constraint

in Figure 5.35, the use of intermediate role Role1B will result in deadlock if that role's hidden selection of yes/no deviates from the choice made by Role2B.

In summary, splicing presents difficulties when an event e in one connector has no equivalent event (or event sequence) in the other connector. When the intermediate roles do not make nondeterministic choices, a role-bridging wrapper may be employed. In other cases, a glue-bridging wrapper may be attempted, but it may prove necessary to use more constrained glues than those provided in the original connectors; this necessity can be determined by using safety properties but its resolution is likely to require greater human intervention than the instantiation and application of a stock template. Finally, for some situations (those in which it is permissible for the non-shared event e to always appear in company with a specific shared event e_s) an acceptable spliced connector *may* be obtained by using an expand/compact pattern rather than an insert/delete pattern. For example, in Figure 5.31 one might expand write into a sequence of open, write, close; the role that is aware of open and close will have more constrained behavior than in its original connector. Safety analyses are key in checking whether behavior is sufficiently constrained; "sufficient" (as stated in a safety property) must be defined by the person using the splicing transformation.

5.5 FSP vs. Wright

In this section, which is optional reading, I discuss some of the differences between FSP and Wright, and their implications.

Wright distinguishes initiator and observer; FSP does not. Similarly, Wright dis-

```
Caller = (hide → call → return → Caller
         | hide → end → END) \{hide}.
```

Figure 5.36: FSP vs. Wright: "Internal choice" via hiding

```
// The client is missing an "init" event.
// We expect this problem to be detected; will it be?
Client = (request → result → Client).
Server = (init → Operate),
   Operate = (request → result → Operate).
Glue = (client.init → server.init → Glue
        client.request → server.request → Glue
        client.result → server.result → Glue).
||Faulty = (client:Client || server:Server || Glue).
```

Figure 5.37: FSP vs. Wright: Can't enforce "external choice"

tinguishes two kinds of choices, internal and external; FSP has one kind of choice. I have shown how to "fake" internal choice in FSP by introducing a "special" event, e.g., hide, in a process that needs to emulate internal choice, and then hiding that event at the end of the process definition. An example of this appears in Figure 5.36.

External choice, however, presents greater difficulties. Consider a simple client-server connector with initialization. In this connector's protocol, the client must command the server to initialize, before alternately issuing requests and receiving results. If the initialization event is left out of the client role, in FSP the glue is able to engage in the event *at will* because client.init appears in no other process's alphabet. Figure 5.37 illustrates this case. LTSA will detect a deadlock, but only because the glue can interleave a second client.init at a point when the server role is unwilling to engage in server.init; the glue cannot make further progress and the connector is deadlocked. In Wright, the glue would not be able to engage in client.init at all (because the choice elements in the glue would be expressed as an external choice between observed events) and the root of the problem, i.e., the client's missing initialization, might be more readily apparent to someone checking for deadlocks.

Wright also has a notion of "successful termination", which cannot be achieved until the processes have all agreed to terminate. In Wright, success (§) is actually

$\surd$ followed by STOP where $\surd$ is a special event. FSP at least has a notion of non-erroneous termination (END, acceptable termination, vs. STOP, bad termination) but would require a convention to achieve the equivalent of $\surd$. For example, I could designate some event end as a "special" event *by convention*; I would have to take care to explicitly strip any prefixes, such as caller., from end in all processes referred to in the final definition of the composite process.

In short, FSP is not designed to describe *some* things that can be detected in Wright; attempts to do so may succeed but result in an accretion of slightly awkward conventions (and, potentially, associated checks for human error, for example to ensure that all prefixes have been stripped from a conventionally "special" event). There is a tradeoff between notational complexity (which may frighten away potential users) and the ability to distinguish additional kinds of protocol design errors. With increasing notational complexity there is also a risk of simply using the notation incorrectly (selecting the wrong choice operator) which also necessitates the ability to perform such additional checks. For a non-Wright-specific discussion of this issue, see section 8.1.2.

5.6 Summary

A formal means for describing connector transformations, independent of the connectors to which they may be applied, can provide useful benefits to those who use connector transformations by supporting the understanding and analysis of the transformations. I describe connectors as structured protocols in FSP, and use this as a basis for describing connector transformations as transformations on these protocols.

By focusing on the protocol facet of a connector and basing my approach on process algebra, I am able to state and analyze properties related to the soundness, transparency, and compositionality of connector transformations. With respect to soundness, it is possible to check whether a transformation introduces deadlock (as well as any event orderings that may be designated as illegal) into a connector, and it's possible to check whether the transformed protocol makes progress; with respect to transparency, it is possible to check that a specific transformation does not change the interface (roles) presented by the connector. Compositional issues, such as the effect on the connector protocol of re-ordering a pair of (perhaps non-commutative) transformations, can be analyzed. In addition, domain-specific analyses (such as, does a purportedly reliability-enhancing transformation successfully mask errors from a client role) can be stated and performed to confirm whether a transformation provides a desired enhancement.

By providing a collection of templates for describing connector transformations as protocol transformations, which may be used to create parameterized domain-specific instances of transformations, I provide a range of leverage and reusable pieces. Specific kinds of analyses that may be desirable for particular connector transformations are suggested in the collection of templates.

Chapter 6

Tools for Implementation Generation

6.1 Introduction

As seen in the preceding chapters, the idea of connector transformations provides a different way of thinking and reasoning about a large monolithic modification to a connector. However, a more compelling contribution of connector transformations is the potential to provide a middle ground between the use of existing connector implementations (which, though convenient, may not meet extrafunctional requirements) and the laborious creation of new connector implementations by hand.

This chapter supports the claim that the use of connector transformations makes it *possible* to generate implementations of complex connectors. Subsequent chapters will confirm that this approach also makes it *easier* to produce such implementations.

The demonstration of the feasibility claim takes the form of a tool I have implemented that operates on elements of a connector implementation, with additional inputs specific to a connector transformation, and produces elements of an augmented connector implementation. The tool makes it straightforward to compose multiple connector transformations to create a more complex augmentation. In this chapter I describe the approach taken in producing such a tool, give an example of its use, and describe the transformations that are currently supported.

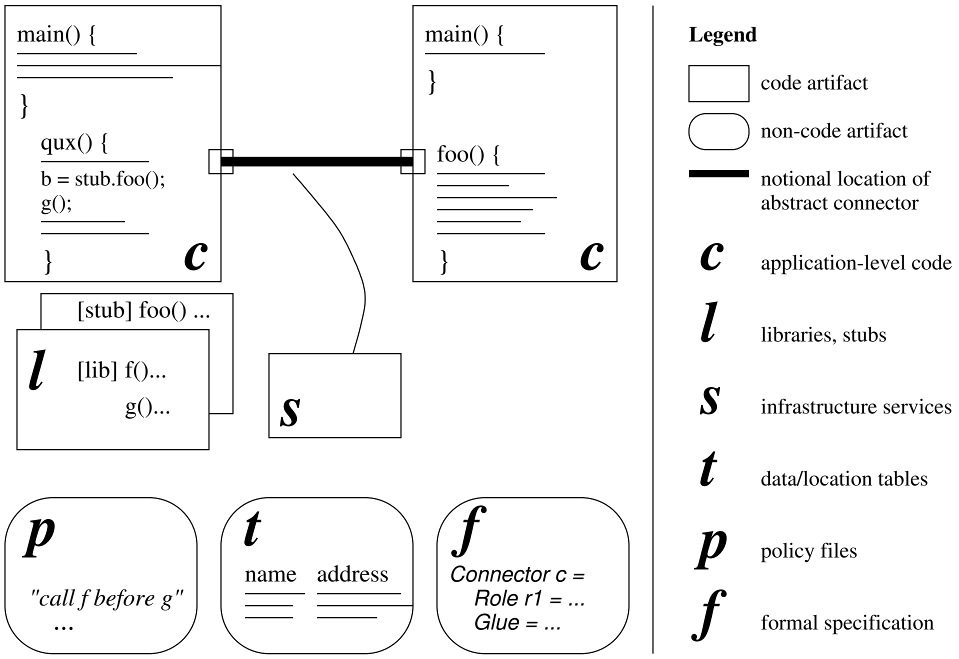

Figure 6.1: Concrete artifacts comprising a connector

6.2 Transforming Connector Implementations

Here I consider the connector transformation tool support in general terms, beginning with a review of the parts of a connector implementation and a brief discussion of what attributes a tool should provide; I then describe an approach intended to meet these criteria and discuss the strengths and limitations of the approach; then in the following section I give an overview of the specific tool that was created using this approach.

6.2.1 Connector Implementations

Figure 6.1 repeats the diagram of connector implementation parts first seen in Chapter 3. Recall that at the level of implementation, a connector is a collection of several kinds of concrete artifacts, some of which may be intermingled with other parts of the system (such as the component implementations). This makes generation or modification of connector mechanisms, whether automatically or by hand, a difficult and nonlocalized task in which multiple locations are affected.

In modern connector technology, it is common to have mechanisms (such as stub

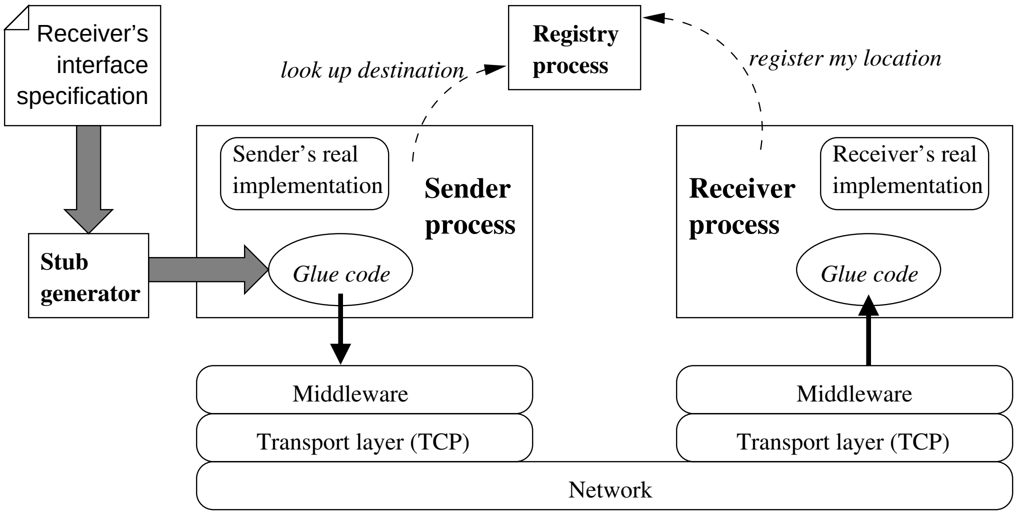

Figure 6.2: Runtime view of common mechanisms

generators or connection factories) that produce proxies, and lookup services that provide references to proxies. A stub generator operates statically on an interface description to produce proxies for a remote component corresponding to the interface description; a connection factory produces proxies for multicasting. The local caller uses a lookup service to obtain a reference that is needed in order to use such a proxy[1]. For example, this might be a reference to a stub for a named remote component, or it might be references to a connection factory and a destination (such as a queue, for point-to-point events, or a subscription topic, for broadcast events).

Figure 6.2 illustrates such a system. Here, a sender process is about to communicate with a receiver process. The sender process includes both the sender's real implementation and automatically generated "glue code" (a stub). This stub was created earlier by giving a specification of the receiver's interface to the middleware's stub generator. (The stub was either placed on the sender's host at that time, or at a location from which the sender is now able to download it.) The receiver process has registered its own location at a "well-known" registry service; here, well-known simply means that the sender process knows how to find the registry service. The sender process has obtained a reference from the registry service (and, if the stub was not placed on the sender's host initially, the sender process also downloads the stub); this reference will enable the middleware to direct the communication to the sender

[1]To save overhead it is common, though not required, for the local caller to obtain (and hang onto) this reference once before its first use, rather than look it up once per use.

process. To communicate with the receiver, the sender's real implementation makes a local call to the glue code. The call traverses a set of layers from the middleware to the network; at the receiver's host, the call goes up from the network to the middleware, and the glue code in the receiver process makes a local call to the receiver's real implementation.

The approach outlined later will leverage these lookup services and proxy generation mechanisms; abstractly, given stublike code on either end of the connection, we have a choice of inserting mechanisms "before" or "after" the stub by modifying and/or generating source code ("in" the stub is also a theoretical option but could negatively impact maintainability), and we may also tinker with the registry services that provide the stub. To some extent therefore it requires base connectors to be built on existing middleware that uses generation technology. This particular tool implementation relies on this assumption, but the assumption is not intrinsic to the overall approach; it would still be possible to provide tool support for a base connector[2] that, perhaps due to being built up idiosyncratically from low-level communication mechanisms, does not have a call that is unambiguously recognizable as a proxy instantiation (the fundamental element of this structure) or a communication library call. This issue is revisited in the Discussion chapter, section 8.6.1.

6.2.2 Desiderata

There are several desirable attributes for a tool that supports application of connector transformations to implementations. These criteria are necessary to support the thesis statement's claims that a compositional generational approach is not only possible but also can be rapid and easy (and produce a wide variety of useful complex connectors).

First, the tool should provide a reasonable breadth of possible results. It should not be inherently limited to a single base connector (able to produce only variants of that connector implementation) nor to a uselessly small number of transformations; support for at least three connectors, and at least three transformations each, is necessary to demonstrate breadth. Also, composition of modifications should be straightforward, so that complex modifications can be constructed.

[2]Or a range of such base connectors, provided that there is sufficient commonality in the low-level mechanisms that they do use; if these mechanisms are provided by a common substratum such as an operating system this may of course be a reasonable expectation.

Second, the tool should support reuse to reduce the cost of developing the implementation of a new augmented connector type[3]. Consider the following scenarios: one, reusing a fully or partially instantiated augmentation on the same base connector type (such as "adding gzip compression to Java RMI"); two, reusing a supported base connector type but applying a new augmentation to it (such as "adding something different to Java RMI"); and, three, augmenting a previously unsupported connector type (such as "adding gzip to JMS, a publish-subscribe connector"). These scenarios represent different levels of potential reuse. The tool should provide time savings compared to hand modification in at least the first two of these scenarios; to support the third scenario, extension of the tool support to additional base connector types should at least be possible, and the time spent may be amortized over the time savings in future instances of the other scenarios.

Third, the tool should provide abstraction away from particular base connector implementations to reduce the amount of low-level guru-like expertise required to create an implementation of an augmented connector type. Such an intimate degree of knowledge of the details of a particular base connector implementation (including auxiliary mechanisms such as stub generators), the communication protocols, and the operating system should not be required in the first two reuse scenarios above.

As I've already argued in Chapter 1, these attributes would provide an improvement over the status quo of ad-hoc modifications made by hand (either to application-level code or to infrastructure mechanisms such as the stub generator) to create a new connector type. Ad-hoc changes made to application-level code are generally not reusable. Changes made to a stub generator (or other infrastructure mechanism) require deep knowledge of its workings and are only slightly more reusable. In either approach, if several modifications must be composed, they are likely to become entangled with one another, making maintenance and any further modifications increasingly painful.

[3] Of course the tool should also save maintenance effort when there is a new release of an existing augmented connector type and an existing augmented system needs to have its connector implementations updated. In this case connector transformations would compare favorably to an alternative approach of hand-modifying the connector implementation. This will correspond to the first or the third scenario, with additional reuse of the system-specific parameter settings also.

6.2.3 Approach

At a high level, the approach I take has two important characteristics.

First, the tool incorporates knowledge of a base connector implementation so that the tool user doesn't have to be a guru in that area. The tool knows how to find abstract locations, such as "initialization of the connector", within software systems that use a specific base connector implementation. I identify a collection of such "hook-in" points[4], described later in Table 6.1. Transformation applications are performed in terms of these abstract locations. This provides a useful degree of separation between connector-specific knowledge and domain-specific knowledge.

Second, code is not intermingled between the successive transformations in a composition. This separation facilitates maintenance: it supports easier traceability of problems to the responsible transformation and enables future deletion or upgrading of individual transformations. In contrast, when several ad-hoc modifications are made to a connector by hand (as discussed above), they are likely to become entangled, and these useful properties are lost. Note that reducing the intermingling of code does not mean that transformations cannot constructively interact or are orthogonal in their effects: although the *code* is separate, the *data* introduced by one transformation may be used by another transformation.

Note that some expertise on the part of the tool user is still required. When creating a new domain-specific augmentation as a composition of supported transformations, the tool user needs to have domain-specific knowledge, as well as non-domain-specific knowledge of how to operate the tool (which will be outlined in the following section). For example, to create a security-enhancing augmentation it is necessary to have knowledge of security, such as: familiarity with commonly used means of authentication, existing packages that provide implementations, and how these libraries are properly employed. The domain-specific augmentation might later be reused by a subsequent tool user with a lesser degree of domain-specific knowledge.

In the remainder of the chapter, I will go into greater detail regarding the specifics of my approach, its strengths and limitations, and an overview of tool use for an example base connector type, which will also cover what the tool user needs to know about tool use.

[4]For a discussion of the relationship to aspect-oriented programming see section 2.4.3 in Related Work.

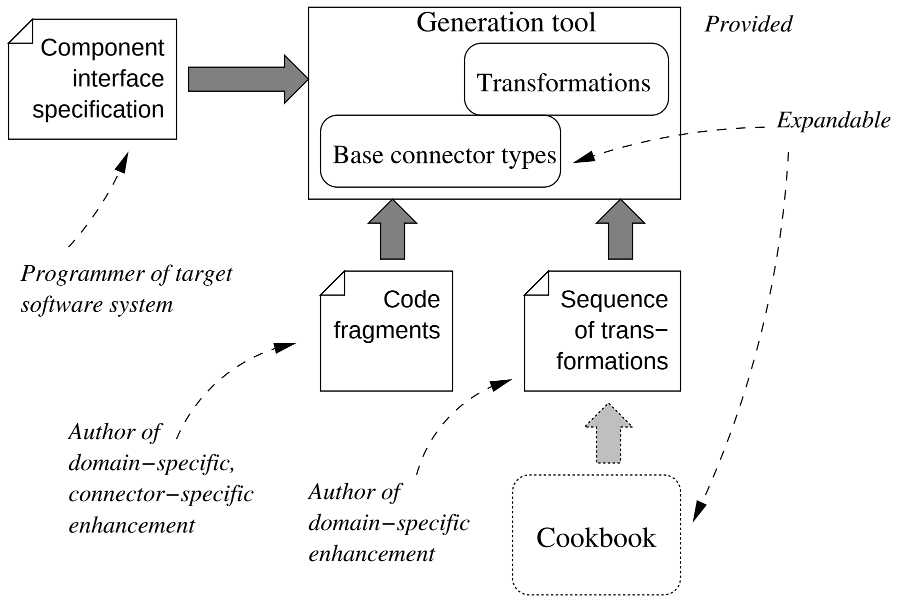

Figure 6.3: Division of labor

6.3 Tool Support

As I will illustrate shortly in section 6.3.3's example, a typical user of this tool will: choose the transformation that is to be used, provide code fragments and other transformation-specific parameters that instantiate the generic transformation as a domain-specific modification, point the tool at an existing connector implementation, and push a button. (Figure 6.3 indicates which tasks are specific to a desired domain-specific enhancement, a target base connector, and a target software system, in order of decreasing generality.) From these inputs, the tool produces an implementation of the new, augmented connector. How does the tool do this? What does it actually produce? I answer these questions here, while also indicating the design decisions that were made in creating the tool: these include decisions to use a chain of proxies, to perform a restricted modification of application source code, and to use code fragments as tool inputs.

6.3.1 Implementation Approach

Transformations are introduced into the connector using the *Proxy pattern* (one of the patterns identified by Gamma et al. [19]). (Figure 6.4 illustrates this pattern with a simple before/after scenario; in the original system, at runtime, "MyClient" has a

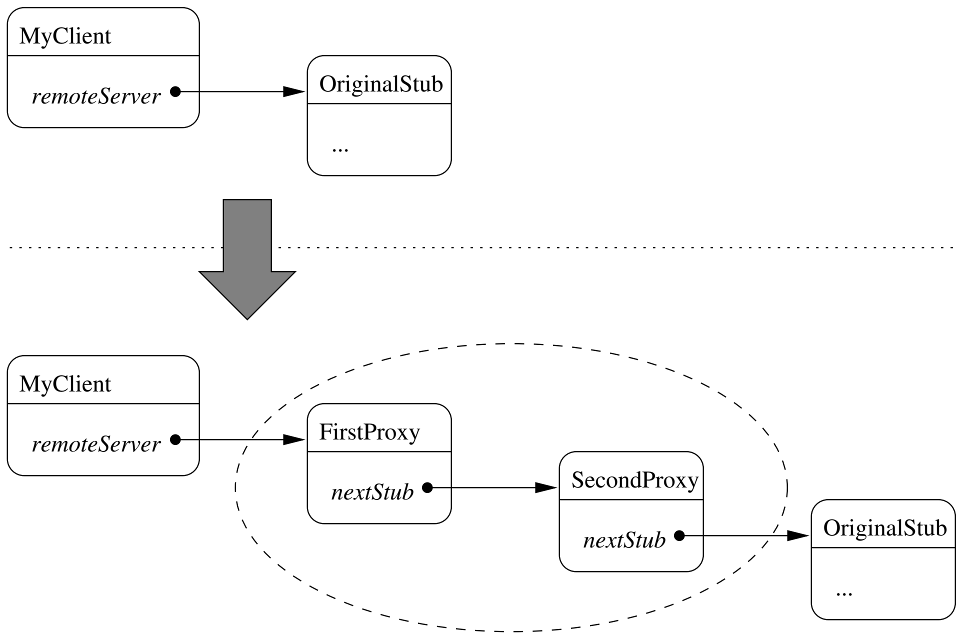

Figure 6.4: A chain containing two proxies

reference to "OriginalStub"; in the modified system, two proxies have been added between them.) As observed in section 6.2.1, the use of lookup services and proxy generation mechanisms in current middleware technology provides an opportunity to place additional proxies between the application-level code and the middleware-provided proxy. A chain of composed transformations produces a chain of proxies. Strengths of this approach include a degree of separation between transformations (as discussed above, avoiding entanglement facilitates maintenance). One potential drawback is inefficiency introduced by the increasing levels of indirection; Chapter 7 will revisit this point with a brief investigation of the performance overhead in a connector generated by this tool. Another potential drawback that is specific to this tool implementation, as section 6.2.1 mentions, is its reliance on automatic recognition, within base connectors, of the lookup/proxy structure of middleware generation technology; this could limit the range of base connectors that this tool can support. A possible alternative to the Proxy pattern is modification of the middleware source code; some inefficiencies (duplication of processing effort) could be avoided in this way with direct access to marshalled data, but such an approach would rely on availability of middleware source code; future work (section 9.3.3) includes addressing the optimizations that are possible when middleware source code *is* available.

The proxy that is closest to a component (e.g., "FirstProxy" in Figure 6.4) is

Abstract Location	Example
Initialization	In a role-addition transformation, we need to obtain a reference to the new component when the system starts up.
Around a communication (before, after, instead, on error)	A role-addition transformation might add a call before, and introduce a conditional around, the communication.
Altering data	A data translation transformation usually alters the communicated data.
Altering response	A data translation transformation may alter the response also. A role-addition transformation may capture (and perhaps modify) the response.
New data members	(A proxy can have data members.)
New methods	(A proxy can have more methods than those required by its interface.)

Table 6.1: Abstract Locations

introduced by performing a one-time *modification* of the application-level source code. This modification is made at specific restricted locations where the application code ordinarily looks up a name and obtains a reference; the tool replaces[5] the lookup call, and returns, in its place, a reference to this proxy. Alternative approaches are possible; one might perform a lower-level intervention by modifying generated code such as stubs or replacing libraries; or one might rely on the middleware's own support for "interceptors"[42]. The chief strength of the application-level source code modification is that it is straightforward and portable; lower-level interventions are less portable (e.g., the format of generated stubs may be subject to change in future releases of the middleware; interception of library calls may be operating system specific) and interceptors are not widely supported at present. However, the drawback of modifying source code is that it relies on availability of application-level source code to modify; this will be revisited in the Discussion chapter, section 8.6.2

The transformation proxies themselves are generated using *code fragments* that

[5]This replacement is related to the previous paragraph's assumption of a base connector with a lookup/proxy structure: we can make the tool automatically recognize the lookup because it is a particular (middleware-specific) library call documented in the middleware's API.

must be provided by the tool user. Examples of these fragments are shown in section 6.3.3 and Figures 6.8–6.11; chiefly, they are code blocks to be inserted into methods of the generated proxy's class. These code fragments make reference to a small abstract set of permissible "hook-in" points (Table 6.1).

The code-fragment scheme just outlined is a compromise between power and assurances; there are other alternatives in the space between the extremes. Here, the tool user may write (or at least include calls to) arbitrary code, but it will only be used at specific points in the communication. An alternative would be to provide a pre-defined non-extensible collection of transformations; such a collection could then include only fragments whose possible interactions have been carefully examined. However, it would also have some of the drawbacks of the existing approaches discussed in section 1.3.

6.3.2 Applying to RMI

To make this more concrete, let's consider how the approach works for RPC-based connectors (in particular, Java RMI). Modern object-oriented RPC implementations provide stub generators and require interface description files for the remote component(s). On the client component's side, we use the remote component's interface specifications as a tool input to generate source code for a proxy that presents the same interface to the client-side code and sits between the client-side calling code and the original stub; we may also make use of the provided stub generator to generate additional stubs. We use the client component source code to generate a new copy of the client component source code, in which the lookup call that would return a reference to the original stub is replaced with a call to a factory method of the new proxy; the proxy's factory method is passed the information necessary to obtain a reference to the original stub. The proxy's factory method, in turn, either calls the factory method of another proxy (if multiple transformations are used) or makes the lookup call (if this is the final link in the chain of proxies). For some transformations, intervention at the client side is sufficient. Other transformations require the creation of a similar proxy at the remote component's side as well; here, rather than hijacking the lookup call, we would replace the call that registers the remote component at the lookup registry.

Using my approach on a Java RMI base connector, the proxy contains a factory method and also implements the methods of a remote interface (see Figure 6.5 for

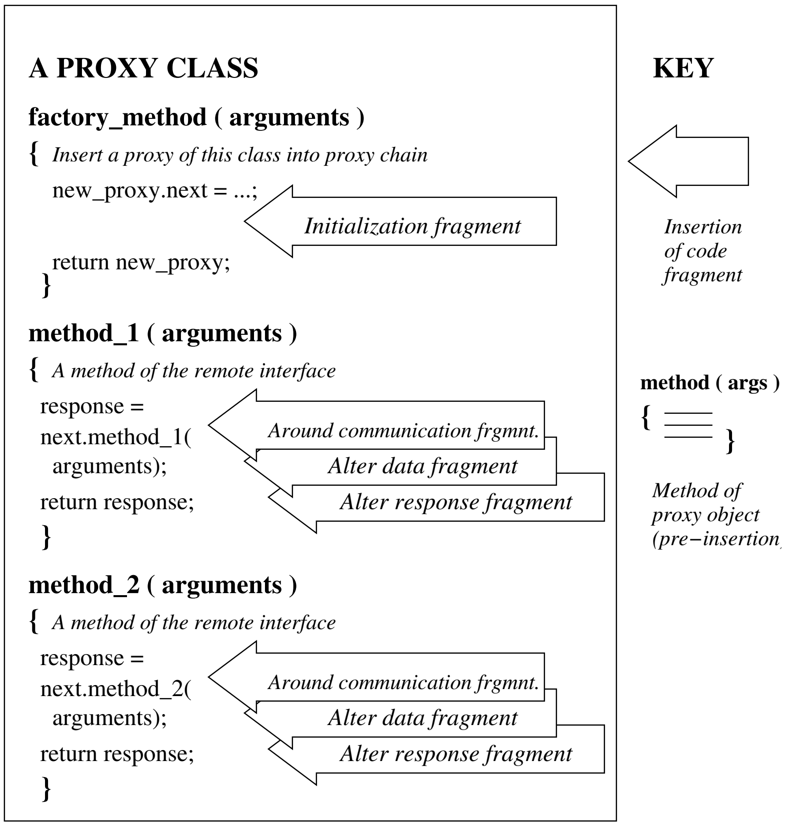

Figure 6.5: Elements of a proxy used for Java RMI

a visual aid); it may also contain additional methods and data members if such are given in the code fragment collection. The initialization code fragment is run when the proxy's factory method is called and is primarily used to obtain a reference to the new component of a role-addition[6] transformation; the factory method also, by default, creates an instance of the proxy and obtains a reference to the next link in the proxy chain. The remaining code fragments occur in each of the remote interface methods (if they are left blank, the method simply calls the corresponding method of the next proxy reference). An altering-data code fragment may access the arguments of the remote call; similarly an altering-response fragment has access to the result (if any) of the call. An around-communication code fragment contains a token representing the location of the remote call; it may thus place code immediately before or after the call, it may direct the call to a different target (which should present the same interface as the original target) or may place a conditional construct or a try/catch construct

[6]The role-addition transformations are *add observer, add redirect, add switch,* and *add parallel.*

around the call. Examples of such constructs are an add-redirect transformation that introduces an authorizer component, which must be consulted to determine whether the call should proceed, as well as a retry or failover augmentation, which catch exceptions thrown by the call.

6.3.3 Simple Example

This example illustrates the application of a simple transformation to a specific supported connector type and indicates what the tool user must understand about the system that is to be modified, what the tool inputs are, and what the tool user does with the result. It also introduces some of the keywords used in constructing the code fragments and shows where one may expect to save time.

For the example system, consider a simple client/server demonstration system in which the clients provide a graphical user interface to "tic tac toe" game boards that are stored on the server; clients submit "move" requests via Java RMI to the server, which is responsible for enforcing the rules of the game, indicating whether a move is legal and whether the game is over. Let's suppose that we are interested in gathering statistics on commonly-used tic tac toe gambits (such as the popularity of a "center" vs. "non-center" opening move). In order to collect this data, we will apply an "add observer" transformation to a Java RMI connector in this system. The architecture of the resulting system is shown in Figure 6.6.

The tool user must find the source code files that contain the server's registration call and the client's lookup call to get a proxy for that server (these may be found by searching for the strings "Naming.lookup" and "Naming.rebind"). In this example, these files are *GameServer.java* and *GameClient.java* respectively. He must also find the source code file containing the server's interface specification. In this example, this file is *GameServerInterface.java.*

The tool user must provide an implementation of an "observer" interface. In this example, the interface is *Observer.java*; the implementation provided by the tool user, which simply displays the observed information in a new window, is *ObserverDisplay.java.* This transformation will run the observer locally on the client's host machine.

In the first phase, the tool is used to generate a client-side proxy, implementing the "observer" transformation. In the second phase, the tool is used to modify the

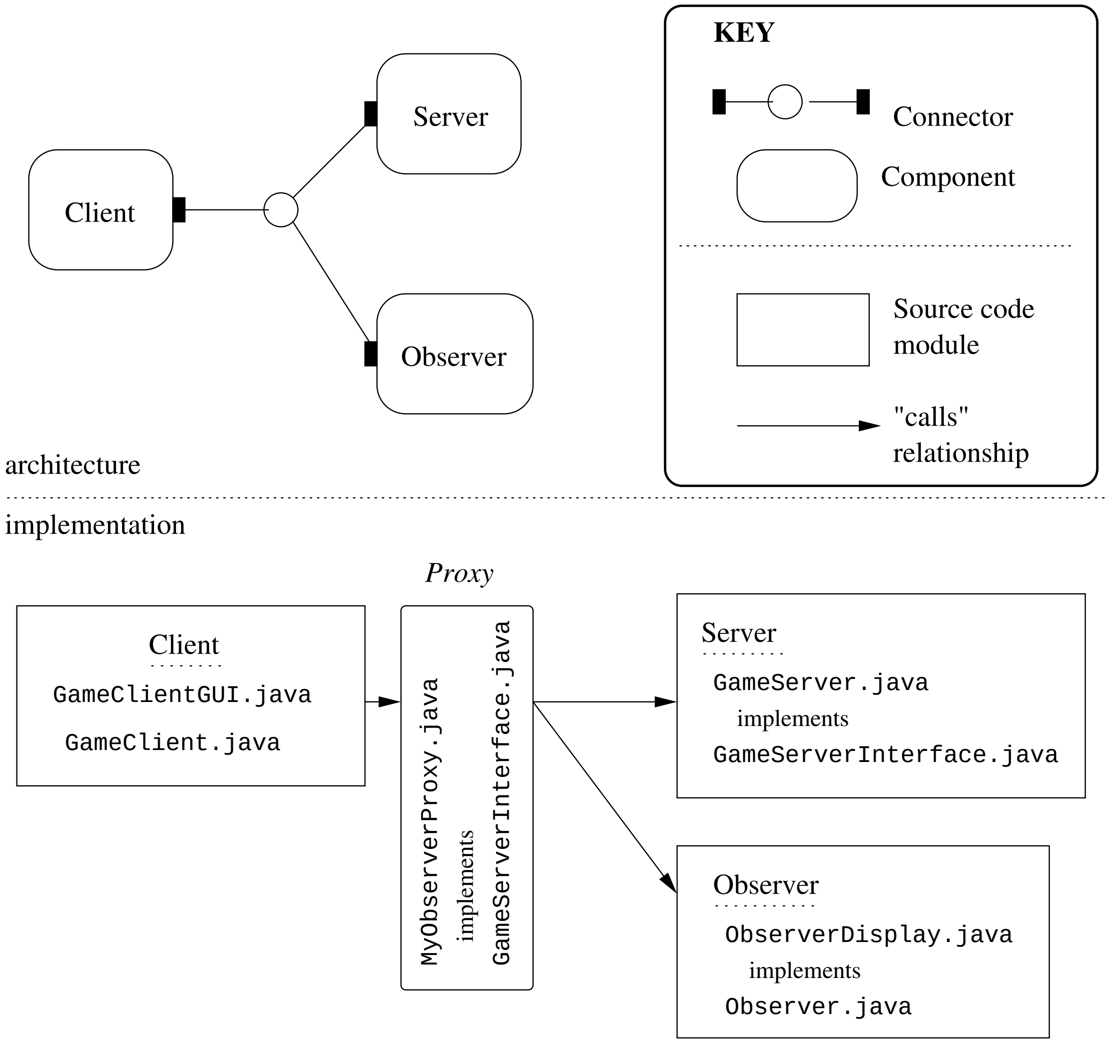

Figure 6.6: A simple client-server system with observer

client's source code to use this proxy.

The tool inputs for the first phase are:

- the location of the remote object's interface file, *full_path/GameServerInterface.java*,
- the name of the interface that will be implemented by the new proxy (which in many cases is the same as the remote object's interface), *GameServerInterface*,
- the role (either "client" or "server") at which the proxy will reside, in this case the *client*,
- the name of the class of the next proxy in the chain, or "end" if this proxy is at the end of the chain and will be calling the generated stub, in this case *end*,

- the constructor call for the added role (when applicable), which here would be *"new ObserverDisplay()"*[7],
- the location of the code fragment collection *full_path/observer1/*,
- and the location where the generated source code file for the new proxy should be placed, *full_path/out/MyObserverProxy.java*.

The contents of the code fragment collection in this example will be discussed shortly.

If the tool user wanted to compose several transformations, the tool would be run several times (with different inputs) in the first phase; the order of generation is not important, since "the name of the class of the next proxy in the chain" (provided as a tool input) determines the order in which the chain of proxies will invoke one another.

The tool inputs for the second phase (in which we modify a line of the client's source code) are much simpler:

- the location of the original client source code file, *full_path/GameClient.java*,
- the location where the tool should place the modified client source code file, *full_path/out/GameClient.java*,
- the name of the remote object's interface file (which will also be the interface presented by the proxy closest to the client), *GameServerInterface*,
- and the name of the class that implements the proxy closest to the client, *MyObserverProxy*.

In the modified client code of this example, the call `server = (GameServerInterface) Naming.lookup("//`*`hostname`*`/MyGameServer")` will be replaced by a call to the new proxy's factory method, passing it the hostname/servicename argument that belonged to Naming.lookup.

Let's take a look at the code fragments the tool user provides that are used by the tool to generate the proxy's source code. First, there is a "new members" fragment that adds two proxy references, one to the "next stub" (which in this example will be the real stub returned by a Naming.lookup call, and which presents the same interface as the proxy currently being generated) and the other to the added "observer" role.

```
// (New Members fragment)
```

[7]To contact a remote observer, the constructor call would be of the form `Observer.factoryMethod("//`*`hostname`*`/MyObserverDisplay")`.

```
private THIS_INTERFACE next_stub;
private Observer extra_stub;
```

Second, there is a "method calls" fragment that assembles the method's arguments into a vector $v1$, calls the observer (*extra_stub*) to report the method call that is about to be made, and indicates that the result of relaying the current call to *next_stub* should be returned.

```
// (Method Calls fragment)
v1 = AVECTOR
extra_stub.call(next_stub.getClass(), METHODNAME, v1);
RESULT:
next_stub.THIS_CALL
```

`AVECTOR` will be replaced by the tool with code that creates a vector $v1$ and adds the method arguments to it in order. `METHODNAME` will be replaced with the name of the method currently being called. `THIS_CALL` will be replaced with the method name and arguments. `RESULT` will capture the result of the *next_stub* call, to be used in the "method results" fragment, if any (if there is no results fragment, the result will simply be returned).

Finally, there is a "method results" fragment that designates a new variable r to capture the result, passes r to the observer, and returns r.

```
// (Method Results fragment)
r = RESULT
extra_stub.result(next_stub.getClass(), r_OBJECT);
return r;
```

`RESULT` is used to stitch together this fragment and the preceding fragment. The `_OBJECT` tag is a request for r to be converted, if necessary, to a type that inherits from Object (e.g., `int` to `Integer`).

These code fragments are used for each method of the interface to create the proxy source code. An example of one such method is shown in Figure 6.7. This method is used to ask the server whether the game represented by *board_id* is over, i.e., a win or draw. The argument is passed to the observer in a vector, the result r of the actual

```
public boolean is_game_over(int board_id) throws RemoteException {
   java.util.Vector v1 = new java.util.Vector(1);
   v1.addElement((Object) (new Integer(board_id)));
   extra_stub.call(next_stub.getClass(), "game_is_over", v1);
   boolean r = next_stub.game_is_over(board_id);
   Boolean r_OBJECT = new Boolean(r);
   extra_stub.result(next_stub.getClass(), r_OBJECT);
   return r;
}
```

Figure 6.7: A method generated from code fragments

call to the server is captured, converted, and passed to the observer (which expects an Object), and finally the result is returned.

These code fragments, though slightly cryptic, need only be written once to generate all the methods of this connector transformation proxy. In this example, the tic tac toe server has five methods (starting a game, ending a game, making a move, requesting the current state of a board, and asking whether a game is over), thus the generated proxy code is significantly longer than the code fragments used to produce it (61 non-blank non-comment lines of code vs. 9 lines).

The remaining tasks for the tool user are to compile and deploy the generated files *GameClient.java* and *MyObserverProxy.java* and the files representing the new observer component *Observer.java* and *ObserverDisplay.java*.

6.3.4 Currently Supported

Having given a specific example for Java RMI, I will now briefly touch on other supported connectors, the differences in supporting *other types* of connectors and connectors in *other languages*, and the connector transformations currently available for the supported connectors.

Java Message Service

Support for Java RMI has already been described[8]. Java Message Service [58], an event-based connector, is also supported. RMI is a remote procedure call connector type. How does the support for an event-based connector type differ?

The approach for this event-based connector is, in the abstract, not very different from the approach used for an RPC-based connector; during initialization, there is a point when participants in the communication register themselves as (potential) listeners and publishers and thus obtain a reference, which we replace with a proxy. This reference can be considered the equivalent, in JMS, of the reference to a *stub* that we saw in RMI. It looks like a local object to publishers; as with an RMI stub, to send out a communication, publishers call one of its methods. Registered listeners provide methods (as with remote objects in RMI) that conform to a particular interface and that are called by the communication infrastructure when a message arrives on the "topic" or "queue" for which they are registered. Let's look at this process in a little more detail.

In JMS, the application source code ordinarily creates an instance of `InitialContext`, which provides access to a lookup service. The application then uses this service to obtain a chain of objects culminating in a Session object. The Session object, in turn, can be used to create either a MessageProducer (which is able to send messages) or a MessageConsumer (which is able to register a MessageListener handler for incoming messages).

To introduce a chain of transformations in JMS, we modify a line in the application source code to replace InitialContext with a subclass of InitialContext; this in turn enables us to replace the Session object with a generated implementation that, when asked to create a MessageProducer or MessageConsumer, returns a proxy. Recall that, for RMI, the tool generated proxies that matched the interface description of the server; here, we generate proxies from the MessageProducer interface description (for transformations located at a "sending" role of the connector) and from the MessageConsumer and MessageListener interface descriptions (for transformations located at a "receiving" role). This subclass of InitialContext is the same for any set of transformations; the generated implementation of Session varies to include the name of the first proxy class in the transformation chain.

[8]A complete list of the transformations implemented for Java RMI is given later in Table 6.2, which summarizes transformation coverage for the supported connectors.

Beyond Java: IPC

Java RMI and Java Message Service are both fundamentally object-oriented. While the implementation generation tool does not take advantage of aspects of the language that are present in Java and would not be present for a C++-based RPC connector (such as garbage collection, dynamic class loading, etc.), recall that it does take advantage of an *object-oriented pattern* to introduce "proxies." For both RMI and JMS, it is assumed that the sender holds an object instance which is standing in for the remote receiver(s); the tool substitutes a *proxy* for this object.

In order to ascertain the applicability of the connector transformation technique to connectors that do not rejoice in an object-oriented language, I have also investigated the adaptation of the implementation generation tool to support "Inter Process Communication" (IPC [53]), a C-based publish-subscribe connector type implemented on top of TCP/IP sockets; it is an outgrowth of the "Task Control Architecture" (TCA) and, like TCA, is intended for use in autonomous systems and distributed task-level control systems [54]. Processes may publish messages, subscribe to messages, or both; messages are routed through a central authority. When requesting to subscribe to messages of a particular type, a process registers a handler function which is called when a message of that type is received. Automatic marshalling and unmarshalling services for messages are provided, but it is the option of each publisher and receiver whether to use them.

Current tool support for modifying this connector includes a subset of the transformations. The primary differences required in the approach to tool support for this C-based connector were due to the sending components' lack of a remote reference (the receiving components register handler functions, which is an action similar to registration of a handler method and requires little more than additional bookkeeping in the generated proxies for the receiver). To intercept "publish" API calls, there are a few choices, including replacing the call in the application source code (individually or through the use of #define macros) or replacing the IPC library itself. Currently the tool support replaces the publish calls, as well replacing the handler functions in the subscribe/unsubscribe calls with a "proxy" handler function, thus enabling interception of messages at sender and receiver. The tool generates transformation-specific C code for the publisher and receiver proxies from code fragments similar to those shown in this chapter. Supported transformations are shown in Table 6.2; data translation at the receiver is not yet working, and support for role-addition transformations is currently limited to roles that present the same "interface" as the existing receiver role(s).

Extending Connector Support

Key elements when extending support to a new base connector are, first, to provide sufficient program understanding to identify and replace the code that obtains the remote reference for the original connector, and second, to be able to generate a proxy that resembles (in its interface) the remote reference. The tool itself is written in Java and makes use of JavaCC and JJTree; a grammar and parser for the language in which systems using the particular base connector are implemented (and for the interface description language, if different) provides a useful starting point.

When considering new base connectors to support, it is also worth discussing what level of complexity of a connector it is reasonable to support. On one hand, tool implementation is more difficult for an extremely complex connector. On the other hand, selecting an extremely low level interaction mechanism means that more transformations must be employed to build up to connectors of a useful degree of complexity. Some time and effort may be saved by selecting an existing connector of sufficient complexity to be already in common usage. This tradeoff is similar to the discussion of appropriate transformation granularity in Chapter 3.

Transformations

Table 6.2 gives an overview of transformations currently supported, per base connector type. A range of interesting transformations are available, but coverage is not currently complete. Since JMS provides unidirectional event-based connectors (messages do not return results), data translation of a return value is not meaningful for this connector. Aggregate is not currently supported for JMS but there is no fundamental reason that prevents it. A restricted form of splice is partially supported for RMI (it can translate between some interfaces that are mismatched in the number of method parameters); a perhaps more interesting form of splice, which would be possible but is not currently supported, would be to provide a small set of variants of a connector that is "Java RMI" at one end and "Java Message Service" at the other.

A tool user can start with these supported generic transformations and write code fragments to create a domain-specific instantiation. However, I have also built

Transformation	Java RMI?	JMS?	IPC?
Data trans.:			
Arguments	√	√	
Return value	√	(N/A)	(N/A)
Add Observer:	√	√	
Add Redirect:	√	√	
Add Switch:	√	√	√
Add Parallel:	√	√	√
Sessionize:	√	√	√
Aggregate:	√		√
Splice:	(limited)		

Table 6.2: Implementation Coverage of Transformations

up a library of such instantiations which, for common uses, provide further cost reduction and time savings when using the tool. The following instantiations of generic transformations can be *reused* by a tool user with little or no modification to apply that particular specialization of a transformation:

- data translations that operate on a vector of objects or on a byte array,
- add-observer transformations that are passed the method name and a vector of the arguments,
- add-redirect transformations that are triggered by return values (e.g., acceptance test),
- add-switch transformations that are triggered by exceptions (e.g., failover),
- add-parallel transformations that wait for all responses before voting,
- add-role transformations that record (and play back) checkpoints and communications since the last checkpoint,
- sessionize transformations that are triggered by exceptions (e.g., retry).

To instantiate and apply these transformations, the existing code fragments can be reused; the effect of transformations that require a component (or class) that implements an appropriate interface, such as the *ObserverDisplay* component in the simple example seen earlier, can be tailored somewhat by providing a different drop-in component implementation (for example, ObserverDisplay might be replaced by a component that records method calls as a representative workload, or that performs crude timing measurements between the call and the return). Some other transformations, such as aggregate, would require writing new code fragments, but should not require modification of the tool itself.

Extending Transformation Support

One can characterize transformations broadly in terms of the abstract "hook-in" locations for which they require support. For example, the observer transformation in the simple example added code both *before* the call and *after* the result was obtained. Table 6.3 illustrates some of these.

A data translation transformation introduces a function implementation (which may be a new method of the proxy or a method of a new object class). The method that implements the translation function is applied to the *arguments* and/or *result* of the remote call.

A role-addition transformation (add observer, add redirect, etc.) introduces a reference to a second stub[9] which may present a different interface than the original stub, or may present the same interface. It is created at *initialization*. A call to the new stub may be made in addition to (either *before* or *after*) the original call, or, if presenting the same interface, may replace the original stub as the *target* of the call. The *result* of the call (ultimately returned by this proxy's method) may be replaced by the result of the added role's call.

An aggregate transformation introduces a second stub (with the same interface as the original stub) and a decision-making apparatus. These are created at *initialization*. The decision-making apparatus is consulted *before* a call, and determines which proxy is to be the *target* of the call; to inform its decisions, calls to the apparatus may also be made *after* a call or *on error*.

A sessionize transformation introduces additional storage for some value which is to be recorded and later replayed. The *arguments* and/or *result* of the call might

[9]The proxy embodying this transformation will thus contain two references to downstream "stubs or proxies." I refer to them here as stubs, to avoid confusion with the proxy embodying *this* transformation.

Transformation	Parameter	Hook-in
Data trans.	Function implementation	New method of proxy, or method of a new class (new data member of proxy)
	...What the function is applied to	Arguments of call and/or result of call
	Location	(Tool input)
Add { *Observer,* *Redirect,* *Switch,* *Parallel* }	New role	A second proxy (new data member of proxy)
	...Created at	Initialization
	Traffic director	Before call or after call or replacing call target
	Traffic set	(Tool input)
Aggregate	Second connector	A second proxy (new data member of proxy)
	...Created at	Initialization
	Selector	New method of proxy, or method of a new class (new data member of proxy)
	Selection points	On error, or before or after call
Sessionize	Recorder	New data member of proxy, or local variable
	When to record	Record call before/after, or record result after call
	Replayer	On error or in place of call

Table 6.3: Hook-ins used by some transformations

be recorded; arguments may be replayed *on error* to retry the call, or in place of arguments of a subsequent call.

6.4 Continuing Example

Chapter 3 introduced a motivating example in which I proposed to augment an RPC connector in two ways.

The first enhancement was for the client to retry requests when an error is returned; because the server's methods are not idempotent, we must also add duplicate detection to make this transparent to the server.

The second enhancement was to add a "backup server" to be used when failure of the primary server is suspected. Because the server is not stateless, we must also add a means of updating the backup server's state when it is called into action.

Each of these enhancements requires more than one transformation; moreover the two enhancements are both to be applied to the same connector. Some care must be taken when composing transformations to achieve the desired outcome, since, as we have seen in Chapter 5, transformations are not necessarily commutative even if they appear orthogonal. (Getting the order right is a matter of a few minutes' thought and perhaps a sketch on the back of an envelope. Getting the order wrong will either be immediately apparent, and may even be detected by the compiler, or will result in a connector that still functions properly but less efficiently; for example, swapping retry and failover results in a connector that may switch to the backup server when the primary server is still functional.) We will chain them in this order, from the client role to the server role: client; add a unique identifier; keep checkpoints; failover; retry; *transmit via the real stub;* check for duplicates; remove the unique identifier. The transformations that add and remove the unique identifier must occur outermost, "outside" of any transformation that may create duplicates (to ensure that duplicates actually have the same identifier and can thus be detected). The transformation that keeps checkpoints must occur "outside" of the failover transformation (which creates a fork to two different servers), so that it can update a new server with the old server's checkpoint. The failover/retry ordering should be familiar from the previous chapter.

Let's consider the code fragments used for these transformations. Note that while they may appear intimidating at first, there are mitigating factors. Users of the tool will not often have to write code fragments entirely "from scratch"; more commonly they will be modifying or reusing existing sets of code fragments, which for similar transformations will be similar in structure. For example, all role-addition transformations introduce an "extra stub" member representing the added role, and will either make calls to both stubs, or use a conditional statement predicated on a "traffic director" object to determine which stub to use.

Figure 6.8 shows the code fragments for a data translation transformation that marshals arguments into a byte array, calls the data translation's *function* to calcu-

New Members fragment

```
private NEXT_INTERFACE next_stub;
private ByteArrayTagger data_translator;
```

Method Calls fragment

```
b1 = MARSHAL
byte[] b1_old = b1;
if (data_translator == null)
  data_translator = new UniqueIDCreator();
byte[] tag = data_translator.createTag(b1_old);
b1 = new byte[b1_old.length + tag.length];
System.arraycopy(tag, 0, b1, 0, tag.length);
System.arraycopy(b1_old, 0, b1, tag.length, b1_old.length);
RESULT:
next_stub.THIS_CALL
```

Method Results fragment

```
b_array = RESULT
real_result = UNMARSHAL
return real_result;
```

```
public class UniqueIDCreator implements ByteArrayTagger {
  int counter;
  private static byte[] toByteArray(long x, byte[] array) {
    for (int i = 0 ; i < array.length ; i++)
      array[i] = (byte) ((x >> (i*8)) % 0xFF);
    return array;
  }
  public UniqueIDCreator() { counter = 0; }
  public byte[] createTag() {
    return toByteArray(counter++, new byte[4]);
  }
}
```

Figure 6.8: Code fragments for Data Translation to prepend a tag

lates a tag, and prepends the tag to the byte array. A tool user who is reusing these code fragments may supply any class that implements a ByteArrayTagger interface. Here I have supplied UniqueIDCreator[10] which creates a simple "unique" identifier. We must also write ByteArrayTagger and UniqueIDCreator and supply their class files.

Figure 6.9 shows the code fragments for an add role transformation that logs requests made since the last checkpoint. The tool will be run twice for this transformation. In the first run, we will direct the tool to apply the first set of code fragments (which add requests to the log, and play back the log on error) to all methods of the remote interface. In the second run, we direct the tool to apply the second set of

[10]In this figure, the initialization of `data_translator` is hardwired into the code fragment, but more generally the class name would be supplied as an argument to the tool to be incorporated in the proxy's initialization code, as seen in the earlier tic-tac-toe Observer example; the code fragments would refer only to the interface name.

New Members fragment

```
private THIS_INTERFACE next_stub;
public Logger extra_stub;
```

Method Calls fragment

```
try {
  RESULT:
  next_stub.THIS_CALL
} catch (RemoteException e) {
  extra_stub.replayLog(next_stub);
  RESULT:
  next_stub.THIS_CALL
} finally {
  extra_stub.recordCall(METHODNAME, arg);
}
```

New Members

```
private THIS_INTERFACE next_stub;
```

Method Calls

```
try {
  RESULT:
  next_stub.THIS_CALL
} finally {
  ((STUB) next_stub).extra_stub.reset();
}
```

Figure 6.9: Code fragments for Add Role to log requests since checkpoint

code fragments (which clear the log) to a specific subset of the interface methods. The Logger implementation is omitted for brevity; it maintains a vector of method call names and their corresponding arguments and makes use of reflection on the `next_stub` argument to (re-)invoke the methods when replaying the log.

In Figure 6.10 there are code fragments for an add-switch transformation. The traffic director is consulted prior to method calls to determine which stub to use, and is kept informed of errors. The simple TrafficDirector I implement here has no provision for failure of the backup or for detecting recovery of the primary. If a RemoteException occurs, the exception is thrown again, because we expect an "upstream" transformation to need to know when the changeover from the original role to the added role has occurred. (Specifically, an exception will trigger the replay behavior of the Logger seen above.)

The code fragments shown in Figure 6.11 are used for a Sessionize transformation that will retry a failed call exactly once. This particular transformation creates a "session" whose duration is only from the time of the original call to the time of the successful result (or the second exception), for a connector in which at most one call

New Members fragment

```
private THIS_INTERFACE next_stub;
private THIS_INTERFACE extra_stub;
private TrafficDirector traffic_director;
private class TrafficDirector {
  private boolean primary_ok;
  public TrafficDirector() { primary_ok = true; }
  public boolean isPrimaryOk() { return primary_ok; }
  public void sawException(Exception e) {
    if (primary_ok)
      primary_ok = false;
  }
}
```

Method Calls fragment

```
if (traffic_director == null)
  traffic_director = new TrafficDirector();
try {
  if (traffic_director.isPrimaryOk() {
    RESULT:
    next_stub.THIS_CALL
  } else {
    RESULT:
    extra_stub.THIS_CALL
  }
} catch (RemoteException e) {
  traffic_director.sawException();
  throw e;
}
```

Figure 6.10: Code fragments for “Add Switch” for failover

New Members fragment

```
private THIS_INTERFACE next_stub;
```

Method Calls fragment

```
try {
  RESULT:
  next_stub.THIS_CALL
} catch (RemoteException e) {
  RESULT:
  next_stub.THIS_CALL
}
```

Figure 6.11: Code fragments for Sessionize for retry

may be outstanding; thus, in this case, we do not need to add any data members for storage external to the local state of the method.

The “function” of the data translation in Figure 6.8, the logger role added in Figure 6.9, and the “traffic director” of the add-switch transformation in Figure 6.10 are all examples of parameters seen in Chapter 3; here, these parameters are embodied by a class provided by the *fragment author* or the *fragment user*. For purposes of illustration I have shown a range of possible alternatives. First, the fragment author may provide a fairly generic interface (e.g., ByteArrayTagger) and *expect* the fragment user to state and supply an implementation of that interface (e.g., UniqueIDCreator); in effect, this "parameter" of the transformation is not (entirely) instantiated yet. Second, the fragment author may provide a class implementation (e.g., Logger) external to the fragment and incidentally *allow* the fragment user to replace the implementation. Third, the fragment author may explicitly incorporate a class implementation (e.g., TrafficDirector) in the fragment; in effect, this "parameter" of the transformation is instantiated and can't be changed by the fragment user[11]. There is a tradeoff between the flexibility of the supplied code fragments and the degree of effort expected of the fragment user.

6.5 Conclusion

This chapter supports the claim that "we can build a transformation-based tool that facilitates the generation of implementations" of new connectors.

First, I have shown it's *possible* to use the tool to generate and compose a useful range of connector transformation implementations. That the tool *facilitates* connector generation, in the sense of making the creation of enhanced connector implementations *faster* (saving time), will be argued in more detail in the next chapter; the simple example tic tac toe server, in which a small number of lines of code are used to generate a significantly larger source code file (9 lines vs. 61, in the example shown earlier in this chapter), is a small step in this direction.

Second, guru-like expertise in the details of a particular base connector is not required to use a connector transformation generation tool; the tool encapsulates knowledge required for proxy introduction. This supports the claim to "facilitate" in the sense of making connector enhancement *easier*. (However, the tool user still needs to be able to recognize a particular base connector: for example, he should be able to tell that a particular distributed Java system is using JMS rather than RMI, or vice versa.) This claim will be reinforced in the next chapter, which describes an experiment in which a novice user of the tool was asked to perform a transformation.

[11] The fragment user can still change this transformation parameter by modifying the code fragments, but this step effectively elevates him to the position of *fragment author* and may require him to have slightly greater expertise in the use of the tool than simply using unmodified sets of code fragments.

Third, it is possible to support multiple connector types, and furthermore the same kind of transformation (such as “add an observer”) can be used on different connector types. In the next chapter I will give further support for this claim of breadth (which is derived from the claim to produce a “wide variety of useful complex connectors”).

Fourth, this approach supports maintenance; it is easy (and localized) to change a small aspect of a particular domain-specific instance of a transformation and regenerate the slightly altered connector, as will be further demonstrated in a case study in the next chapter.

Finally, the nature of the tool support incidentally decreases the opportunity for programmer error as compared to modification by hand. As seen in the simple example of the tic tac toe server, the several methods of a connector transformation proxy are generated from *one* set of code fragments; let’s say that this provides the tool user with one “opportunity” to introduce bugs. If the n methods in the proxy code had been hand-written (e.g., using a “cut and paste and modify” approach) rather than generated, there would be n opportunities to introduce bugs, since each method would provide an independent opportunity.

Chapter 7

Validation

Let's review the three thesis claims introduced in section 1.4.2: "We can define a small set of basic transformations that, when applied compositionally to simple communications-based connectors, produce a wide variety of useful complex connectors. These transformations can also be given a formal interpretation that allows us to understand their properties with respect to communication protocols. Furthermore, we can build a transformation-based tool that facilitates the generation of implementations of these new connectors."

Previous chapters have already supported several parts of these claims. Chapter 3 gave a definition of "a small set of basic transformations." Chapter 5 presented "a formal interpretation" and showed how to analyze some protocol-related properties using this interpretation. Chapter 6 discussed the design, implementation, and use of "a transformation-based tool" to generate new connector implementations by applying compositions of transformations to simple communications-based connectors.

This chapter describes the validation of the remaining adjectives: that, by composing and applying transformations to generate new enhanced connectors, it is possible to arrive at a *wide variety* of *useful* complex connectors. This claim has two aspects: demonstrating breadth and demonstrating utility.

In order to demonstrate breadth, I show that it is possible to produce several kinds of domain-specific modifications by composing connector transformations and that it is possible to apply transformations to several kinds of base connector types. In this chapter, I begin by describing case studies in which a set of modifications is applied to a base connector in an existing and interesting software system.

To demonstrate utility of the technique, I have examined several aspects of what it means to be useful. First, the enhancements should be ones that have been used in the past; in selecting the domain-specific modifications for the case studies, I have chosen techniques that are *well-known*[1] in the domain. Second, the base connectors should also be in use; for the case studies, I have selected base connectors that have been used in existing software systems. Third, the process of using connector transformations to create enhanced connectors should require less effort than modification by hand; I provide a comparative task analysis and describe an experiment in which an inexperienced user of the generation tool modified a connector. Fourth, the tool-generated implementations should be usable in terms of performance; I compare a generated implementation to a hand-modified implementation to determine how much overhead may have been introduced.

Each case study description has the following structure: a description of the software system and the base connector type; a list of the modifications and how they have been composed from transformations; and any interesting results including lessons learned, predictions, or surprises.

7.1 Initial Experiment: Kerberos

During early development of the tool, I used connector transformations to add Kerberos [43] authentication to Java RMI connectors. The purpose of this experiment was to investigate the feasibility of the connector transformation approach, with respect to the production of new and modified source code.

To add the steps of the Kerberos protocol (version 5) to a Java RMI connector, I used three connector transformations.

Add Redirect - This transformation adds another party to the communication and redirects some messages to it. In this case the added role represents a *trusted third party* whose knowledge will enable a check that the messages exchanged by the original parties are ok. (Adding a trusted third party to a communication would be a common occurence in security.)

[1]It should be emphasized that this work does not endeavor to contribute new dependability-enhancement *techniques*. Rather, the goal is to help a system architect to employ the domain-specific techniques that *already exist*, via rapid, easy development of enhanced connectors.

Data Translation - This transformation performs some operation on the data sent in the communication; in this case it incorporates new information that is needed to prove the sender's identity. (Adding a shared "secret" to a communication, e.g. via cryptographic checksums, is another step that one might expect to see frequently in the context of identifying a communicating party.)

Sessionize - This transformation creates and initializes additional state; in this case, it is needed to store the *credentials* that are required by the other two transformations. (Security protocols in which a third party is contacted will often mitigate the overhead of doing so by caching the information obtained; again, this use of the the sessionize transformation is one that is likely to appear in the context of security enhancements.)

To make Kerberos library calls from the Java RMI connector, I used the JGSS[2] package, a Java implementation of the Generic Security Service API; it provides access to the Kerberos V5 libraries (written in C) for Java programs. By examining the examples distributed with JGSS, I determined the JGSS method calls that must be incorporated by each of the transformations (as well as any new variables the method calls would require), resulting in a set of small code fragments. The tool takes care of inserting these fragments in locations appropriate to the connector type and the transformation; for example, the sessionize transformation must add initialization code to be called when the client and remote objects are created, while the data translation's additions come into play later when the remote object's methods are invoked.

In this initial experiment, after some time spent learning about Kerberos itself, it took about two days to perform the decomposition into connector transformations, study JGSS documentation and examples and write the code fragments, and lastly produce acceptable output with the tool (this generated code was inspected, compiled, and run, but did not undergo rigorous testing). The projected savings in terms of lines of code that were hand-written (i.e. code fragments) appeared reasonable as compared to two other possible approaches. First, as compared to hand-modifying the affected *software components* that communicated via Java RMI, with connector transformations each code fragment was written once and then automatically applied to *all* methods exported by the remote object; the alternative would have required writing (and inserting) similar amounts of code for each method. Second, another

[2]Available from the University of Illinois Systems Software Research Group.

option would be hand-modifying the Java RMI *stub generator*, so that it always produces stubs that use Kerberos, but an examination of the sun.rmi.rmic package indicated that the additions required would affect at least four (of nine) classes internal to the stub generator, and would require modification of ten or more existing methods, representing a not insignificant undertaking.

This simple early result suggested that the connector transformation approach was worth pursuing. After further development of the generation tool, I undertook the case studies described in the remainder of this chapter.

7.2 Case Study: Java RMI and VisAD

In the first case study, I modified Java RMI connectors in collaborative VisAD applications.

7.2.1 What's VisAD?

VisAD, or "Visualization for Algorithm Development", is a freely available Java library that enables interactive visualization of numerical data. Two examples of applications that use VisAD are "Galaxy", which allows astronomers to manipulate a simulation of the Milky Way and view its appearance from Earth, and "GoesCollaboration", which allows atmospheric physicists to manipulate weather satellite data [23].

VisAD provides support for *collaborative* visualization applications. In such applications, several users can view and interact with the same scientific data, and changes made by any user are immediately visible to all users. These collaborative applications have a client/server architecture. The client(s) and server communicate using Java RMI[3]; to initiate communication, the server must create and register a RemoteService object (provided by the VisAD library), and the client must look it up in the standard RMI registry.

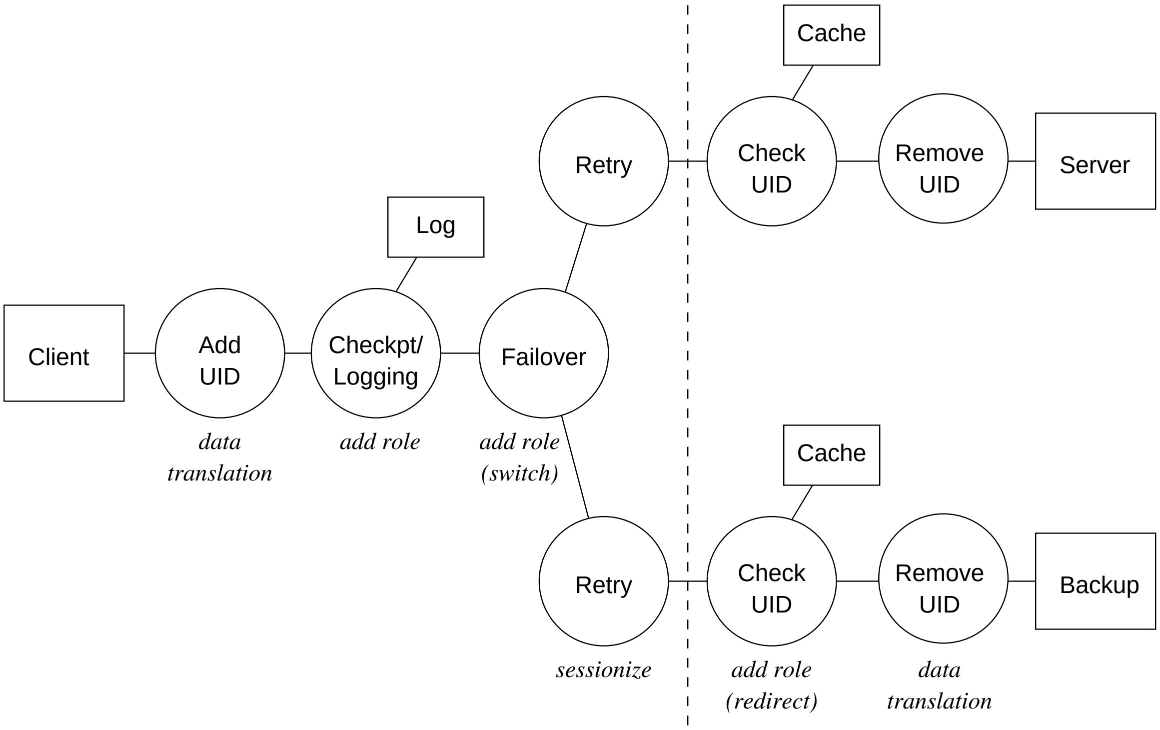

Figure 7.1: Ordering of transformations

7.2.2 Modifications

Suppose that we would like a VisAD-based system to be more resilient to server crashes and to high-variance communication latency. The raw Java RMI support provided in the VisAD library does not have facilities for diagnosing and dealing with situations in which the client is (briefly or otherwise) unable to contact the server. Let's consider the set of well-known dependability-enhancing techniques discussed in Chapter 3 and select a combination of enhancements that could achieve these specific requirements.

To deal with the potential failure of the server, we would like to use a connector that transparently incorporates *failover* to a backup server. Since servers in a VisAD-based system generally have state, we will also need some means of bringing the backup server's state up-to-date before it is called into service; we might do this by maintaining *checkpoints* and/or logs for the primary server and replaying them to the backup server. Finally, because we expect that, in the final deployment environment

[3]For a review of Java RMI, see Chapter 6.

of the proposed VisAD-based system, sometimes a timeout might be due simply to a brief spike in communication latency or a soft (transient) failure of the transmission channel (rather than actual server failure), we would also like to have the connector automatically *retry* any request that times out before resorting to the backup server.

For this case study, I composed the above set of modifications and created an enhanced RMI-based connector; I used the tool support to generate the connector implementation, and tested it in an existing collaborative VisAD system. Let's consider the modifications, and their interactions, in more detail.

- Retry with duplicate detection; this is composed of two simpler modifications:
  - Naive retry. If a request times out, send it again; perhaps this time it will get a response. (Since the fault could have been in the transmission of either the request *or the response*, this could result in sending duplicate requests to the server. Thus it assumes that all services are idempotent.) I use a *sessionize* transformation.
  - Duplicate detection, to protect non-idempotent services. Recent requests (and corresponding responses) are cached at the server role; if a request is repeated, the cached response is returned. (This requires some means of uniquely identifying a request.) I use a *data translation* transformation to introduce a unique identifier, and an *add redirect* transformation to redirect incoming requests to the cache of recent requests.
- Failover. If the primary server is unresponsive, redirect the client's requests to a backup server. (This particular failover modification assumes that the backup server is stateless or up-to-date; the purpose of the next modification given here, logging/playback, is to make the backup server's state up-to-date.) In this composition of enhancements, if "retry" doesn't obtain a response, then we'll consider the server unresponsive. I use an *add switch* transformation to add the backup server.
- Logging/playback. This is a variant of checkpoint/rollback. Client requests are logged (between checkpoints, if any). The log may be played back or cleared. In this composition of enhancements, we'll use playback to update the backup server[4] before switching from the unresponsive primary to the backup. The

[4]The logging approach assumes that all state changes to the server are reflected in, and/or result

failover modification provides the diagnosis[5] that playback is needed. I use an *add redirect* transformation, similar to the form used for a cache, to add the logger.

The transformations used for these modifications interact in two ways. First, some modifications perform "diagnosis" for subsequent modifications: if retry is not successful in masking a problem, for example, then it's reasonable to assume that the server is unresponsive and that it is appropriate to try failover. This manner of cascading several techniques is common in dependability. Second, some transformations may operate on data that was introduced by another transformation: the unique identifier used by the duplicate-caching role was added to the communication by a data translation. The overall transformation ordering, shown in Figure 7.1, follows directly from these interactions: first, cascading modifications must be ordered so that, if X is subsumed by Y or provides a diagnosis for Y, X must be tried before Y; and second, the unique identifier data must be introduced before (and used after) all modifications that re-send a request. The task of ordering the transformations will be intuitive for someone who understands the domain, but is not time consuming (though it may require a little thought) even for someone who is not a domain expert and is reusing existing domain-specific transformations.

7.2.3 Outcome

I began by writing the code fragments for this set of transformations and testing them on a demo system that I wrote earlier (the tic tac toe system of the previous chapter). Then I attempted to apply these transformations to one of the sample applications distributed with VisAD. This process should have been straightforward; however, at this point some minor bugs in the tool became apparent that had not been exercised by demo applications I wrote myself. After fixing the bugs, it took under 30 minutes to apply the transformations, compile, and run the working modified system. For

from, communication. Further, placing the log at the client may be undesirable in the case of multiple clients; an alternative approach, using a different connector transformation, would be to make the backup server an "observer" of the traffic to the primary.

[5] The failover modification, to perform its own proper function, must detect nonresponsiveness of the primary server; since this diagnosis may be of interest to other cascaded modifications that need to take action in the case of server failure, the failover modification described in this chapter will always relay it upward.

this system, the tool generated 962 non-blank non-comment lines of code. Then I applied the same transformations to another of the sample applications distributed with VisAD; this took under 10 minutes (again, generated code was about 1 KLOC), and no further problems were discovered. Section 7.4 will compare these times to a traditional approach.

As an end-to-end check of whether the transformations operated as advertised, I inserted (in turn) each of three kinds of faults that the augmented connector was supposed to be able to mask: dropping a request, dropping a response, and failure of the primary server. To perform the first two tests I added a trivial amount of test harness code to the original base connector to inject the faults on command, and to monitor the communication (viewing the number of retries, etc., in another window); then to perform the third test I killed the primary server's process while the system was running. In the first two tests, the transformed connectors successfully masked the fault by retrying the request. The third test was also successful: the failed request was first retried, which also failed (as it should, given the death of the primary server), and then the request was sent to the backup server which took over and allowed the system to continue operation.

I also investigated the time required to make minor changes to the connector transformations. First, I altered the call-logging policy so that outgoing calls are recorded before their results are returned (rather than waiting for the result). This took 5 minutes including recompilation of the tool-generated code. Next I altered the "retry" transformation to retry a timed-out call up to three times (rather than only once) if it continues to generate an exception. This task took 2 minutes including recompilation. The structure of generated code, in the approach I have taken with this tool, is (as described previously) a collection of cascaded proxies; this results in more layering than a conventional hand-coding approach, in which a programmer might choose to mingle all of the various concerns of the separate transformations to avoid calling through multiple wrappers. (In section 7.5 I describe a simple measurement of the difference in performance overhead of the two approaches.)

7.3 Case Study: JMS and the J2EE Pet Store

In the second case study, I modified Java Message Service connectors in Sun's standard web-commerce demo application, the "Pet Store."

7.3.1 What's the J2EE Pet Store?

To showcase J2EE's (Java 2 Enterprise Edition) support for e-commerce applications, Sun provides a fully-functional web-commerce application [60]; the hypothetical goods being purchased in this application are pets, thus they named this application the Pet Store.

The Pet Store is provided as part of Sun's "J2EE Blueprints" program and demonstrates patterns, frameworks, architectural recommendations, and best-practice guidelines that Sun considers appropriate for scalable e-business applications. As such, it's a reasonable representative of a J2EE application.

The application as a whole is designed around a MVC (model view controller) framework. It includes a web site, a catalog of available pets, and an order processing center. The business logic is encapsulated in "Enterprise Beans." Some communication in this system uses synchronous connectors (such as the queries from the web site to the catalog component); other communication is asynchronous (such as the orders sent from the web site to the order processing center), via Java Message Service connectors. JMS connectors can be either queue-based (asynchronous point-to-point) or topic-based (asynchronous publish/subscribe); both kinds are present in the Pet Store system.

To communicate via a JMS connector, the sender uses a service locator to obtain, by name, a reference to a specific queue or topic; the sender constructs a message and sends the message to that queue or topic. In the Pet Store system, the receivers are "Message Driven Beans" which have been registered to listen to a particular queue or topic (the actual registration is performed by the bean's "container" and is not visible in the bean's source code); the receiver provides a handler method which will be called when a message arrives in the queue or topic. It is generally assumed (but not enforced) in JMS that a queue has only one registered receiver; a topic might have multiple listeners.

7.3.2 Modifications

Suppose that we are concerned that purchase orders may be corrupted in transit. An assortment of enhancements are available to detect and recover the original text. A forward error correction code can be used to increase the redundancy of a message before it is transmitted and (within the limitations of the particular code) to recon-

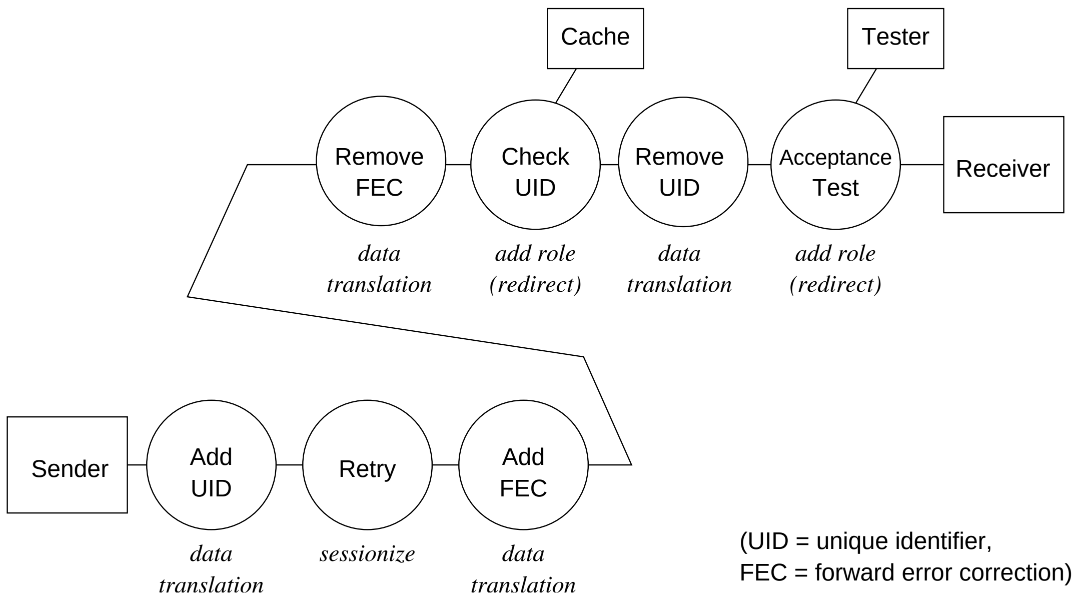

Figure 7.2: Ordering of transformations

struct the original message when it is received. We could also use an acceptance test at the receiver to make sure that the text of the purchase order "looks like" XML syntax, and reject it if it doesn't. In this case study, I added these two modifications, as well as the "retry" enhancement seen in the previous case study.

- Forward error correction (FEC). This enhancement uses a data translation at the sender and the receiver.

- Acceptance test on the received text of a message. For this modification, I use an add-redirect transformation; traffic seen at the receiver is directed to an added role (which performs the acceptance test) and continues to the receiver if and only if it is acceptable.

- Retry. This modification should be familiar. Unlike the procedure-call semantics of RMI, with JMS the sending component does not receive an explicit "result", but it may still receive synchronous notification of some kinds of communication failures; a "retry" transformation can intercept these and attempt to re-issue the original message.

A finite-retry facility offers a possible compromise between the two delivery modes ("PERSISTENT" and "NON_PERSISTENT") offered by an unmodified JMS connector. PERSISTENT guarantees exactly-once delivery from the sender to the destination[6] but has comparatively high overhead. NON_PERSISTENT is a low-overhead option that does not log messages to stable storage; it provides only an at-most-once guarantee. Retry is also potentially useful for "temporary topics," for which only the NON_PERSISTENT mode is available.

This set of transformations seems orthogonal at first glance, but a little care with ordering is still required: whenever a *data translation* and a role-addition transformation (*add observer, add redirect,* etc.) are combined, one must take care that the data received by the added role is in the format that role expects. In this case, the added role does not expect data that incorporates an error correction code, so the role should be added "after" the FEC has been removed at the receiver and the data has been restored to its original format. The transformation ordering is shown in Figure 7.2.

7.3.3 Outcome

Because the receivers' bean containers perform their registration as message listeners, the process of introducing proxies at the receiving role of the connector is slightly different (for this system and, generally speaking, for other JMS-using systems that employ bean containers) than that implied in section 6.3.4. Recall that for Java RMI, the tool modified the remote server's source code slightly so that, instead of registering the remote server, a tool-generated *proxy* is registered. For JMS, the tool still makes a slight modification to the receiver's source code, but rather than replacing the message listener registration (the approximate equivalent, for this connector type, of RMI registration), the tool modifies the receiver's implementation of the message handler method to introduce the call to the tool-generated proxy. When JMS is used without bean containers, the message listener registration is explicit in the receiver source code and it's possible for the tool to replace it with a proxy; for these purposes, either approach has the same effect (in terms of ability to provide the "hook-in" points discussed in Chapter 6).

For this case study, I tested the new transformations on a simple JMS publish/subscribe demo system (provided by Sun in the JMS tutorial) that does not

[6] There is one more leg from the destination to the consumer; whether messages can be dropped on the final leg, due to storage space constraints or other reasons, depends on the message retention policy of the destination.

use Enterprise Java Beans. I then applied the transformations to the Pet Store system. The most time-consuming task was to locate the purchase-order connector in the system source code.

The retry enhancement was reused, with little or no modification from its instantiation in the RMI case study, and required only 10 minutes to incorporate in the simple demo system (including debugging). The other two enhancements, FEC and acceptance-test, used transformations that are supported but that had not yet been instantiated; that is, I had to write or modify code fragments and drop-in helper classes (FEC, as a data translation, requires a translation function, and acceptance-test, as an add-redirect transformation, requires a new component for the added role). These two took about 2.5 hours to instantiate, including debugging and testing in the simple demo system.

Applying this set of transformations to the purchase-order connector in the Pet Store took about 2.5 hours, including compiling and testing the system. This is longer than the VisAD case study, partly because I initially misidentified some parts of the original purchase-order connector (including file names and identifiers that are required as inputs to the tool), and partly because the Pet Store system by nature takes substantially longer to recompile, deploy, launch, and test than the VisAD-based systems. Debugging required revising the filename inputs to the connector transformation tool, to correctly indicate the pieces of the original purchase-order connector; no revisions to the code fragments or drop-in classes, as used on the simple JMS demo system, were needed.

After testing the purchase-order connector, I also applied the transformations to a second JMS connector; this connector is responsible for communication between the order processing component and the order approval component. It took less than 5 minutes to apply the transformations to this connector, plus 15 minutes to rebuild the system and confirm that both connectors were correctly modified.

Finally, as with the previous case study's end-to-end check, I injected errors to test that the intended enhancements in the connectors really were operating. The retry enhancement was exercised as in the previous case study by added test harness code in the pre-enhancement connector to enable me to drop requests on comment and view the actual communication (to determine whether and how many retries were being made). To test the FEC enhancement, I added test harness code to flip a bit in the message communicated by the pre-enhancement connector (thus the message

Task	Best case	Average case	Worst case
Traditional approach:			
Understand connector type	$\leq$ 1 day	1 week	$\geq$ 2 weeks
Understand system	$\leq$ 1 day	1 week	$\geq$ 1 month
Understand change	$\leq$ 1 day	1 week	$\geq$ 1 month
Modify code	$\leq$ 1 day	2 days – 1 week	$\geq$ 1 week
My approach:			
Understand connector type	$\leq$ 1 day	1 day – 1 week	$\geq$ 1-2 weeks
Understand system	$\leq$ 1 day	1 week	$\geq$ 1 month
Understand change	$\leq$ 1 day	1 week	$\geq$ 1 month
Select transformations	30 minutes	30 minutes – 1 hour	1-4 hours
Code fragments	(provided)	30 minutes – 2 hours	1-3 hours
Drop-in code	(provided)	1 hour – 1 day	$\geq$ 1 day
Tool parameters	10–30 minutes	10 minutes – 1 hour	10 minutes – 1 hour

Table 7.1: Estimated task breakdown

is "corrupted", in the enhanced connector, between the time that the checksum is added and the time that it is removed), and to observe communication at this point and at the receiver. I ascertained, firstly, that the corruption *was* actually occuring and secondly that the FEC decoder was successfully *masking* it thanks to the added redundancy. To test the acceptance test, I used similar test harness code which altered the XML text message prior to transmission so that it would provoke a failure of the acceptance test at the receiver, and I observed the receiver traffic (pre- and post-wrapper) to make sure that the message was both received and rejected.

7.4 Task Analysis

The problem of modifying the connectors in a software system can be broken into a set of tasks. By comparing these tasks for a "traditional" hand-modification approach and a connector transformation approach, we can gain a better understanding of the difference in effort between the approaches. In this section I characterize each task in terms of a best-case, average, and worst-case scenario. The time required for a task will vary depending on the scenario. Time estimates are summarized in Table 7.1.

Suppose that a person X plans to modify the communication in a particular software system to provide enhancement of a particular extrafunctional property.

In the traditional approach, X must understand the connector type to be enhanced so that he can modify it, understand the software system that is to be modified and locate connector instances within it, understand the kind of change that is necessary, and modify multiple locations in the source code. In my approach, X must understand the connector type so that he can locate instances of it, understand the software system that is to be modified and locate connector instances within it, understand the kind of change that is necessary and decompose it into connector transformations, provide code fragments to the implementation generation tool, and provide any drop-in code required by the specific transformation.

The first task, in either case, is to *understand the connector type*. However, the level of understanding required will be slightly higher in the traditional approach.

Best case: X is familiar with the use of the connector type that is to be modified, and has modified it in the past. For example, X has used Java RMI extensively and has prior experience adding minor enhancements to it. X is ready.

Average case: X is able to recognize instances of the use of the connector type, through simple pattern-matching on the API calls. For example, X has read Sun's tutorial on Java RMI and has successfully compiled and deployed a trivial example system; X is able to identify an interface file as belonging to a remote object or not and to realize that a source file calling "Naming.lookup" represents a client obtaining a remote reference and that a source file calling "Naming.rebind" represents a remote object registering itself. In my approach, this may be a *sufficient* level of knowledge for some kinds of domain-specific transformations. In the traditional approach, X will probably need to learn a little more than this; for example, he may need to be able to write a new interface file. X may spend on the order of a week becoming more familiar with the connector type.

Worst case: X has not seen this connector type before. X will take a considerable amount of time to achieve the initial level of knowledge of the average case scenario (plus any time required in that scenario).

The next task is to *understand the software system*. Assume that X now has a sufficient understanding of the connector type. Before X can modify instances of this connector within the system, it is necessary to discover where the connector instances are, what components are communicating, and what is being communicated. (The Pet Store case study demonstrates the impact on later tasks of an incorrect initial understanding of the system.)

Best case: X is familiar with the system architecture and implementation. The architecture is documented, including what connector types have been used where, and is up-to-date. X will probably take less than a day to locate, within the implementation, the connector instances.

Average case: There is partial documentation of system structure, and X may be familiar with some parts of the system. X will need to perform basic pattern-matching to find instances of the known connector type. X may take about a week to become more familiar with the system, discover undocumented locations of connector instances, and figure out what communication they are supporting.

Worst case: None of the above. X is confronted with an unfamiliar, undocumented system (or with profoundly incorrect documentation). X will have to spend substantial time in reverse-engineering the system structure (if any), determining the connector types that are in use, and discovering their locations. This time will vary with the size and complexity of the system.

The next task is to *understand the modification needed*. X has figured out where the connectors are in the software system, but he still needs to understand what the new extrafunctional requirement is, what kind of modification could be used to achieve it, and which of the connectors should be modified in this way. For example, if the domain of the extrafunctional requirement is dependability, X will need to understand common *dependability* enhancements and what problems they address.

Best case: The requirement is well defined (e.g., "requests from this component to this server are sometimes corrupted or dropped; make it stop.") It is clear which connector(s) are likely to be involved. X has experience in this extrafunctional domain and knows how to achieve the requirement.

Average case: The requirement may need to be nailed down a little more (e.g., "this component seems to be getting bad data from somewhere."), and X may need to figure out which connector instances are relevant. X has slight knowledge of this extrafunctional domain. X might be helped by a "cookbook" that matches requirements to well-known domain-specific enhancements (e.g., "the following kinds of errors can be masked by using an error correction code.")

Worst case: The requirement may be ill-defined, or X may have been given multiple conflicting requirements; this will require additional time to resolve. X has no practical knowledge of the domain and will need to do some background reading to achieve the level of knowledge in the average case.

The next task, given understanding of the system and the modification required, is to produce an *implementation* of the modification. In the traditional approach (of which System B in section 7.5 is a trivial example), X will be hand-modifying existing source code of the application in multiple locations. The time required will increase with (to use the vocabulary of RMI) the number of methods in the remote interface and the number of call sites.

Best case, traditional approach: The interface is narrow and there are few call sites. X has performed the same modification on another system and may be able to apply a partially cut-and-paste based approach. For a rough comparison, the interfaces in the VisAD case study required about 1 KLOC of tool-generated code; on a fairly productive day, a programmer may produce .5–1 KLOC of hand-written code (if the code does not require deep thought).

Average case, traditional approach: X has not performed this modification before. X knows where to find available resources for implementing the modification (e.g., libraries that implement an error correction code), but is not familiar with their use.

Worst case, traditional approach: There is a wide interface and many call sites; X will have to write more code, in more locations. X has not performed this modification before and cannot find any reusable code or libraries. X may, for example, have to implement a facility for calculating error correction codes himself.

In my approach, to produce an implementation, X must break the modification into transformations. Then X must provide parameters, code fragments, and any drop-in code required by the specific transformations, in order to run the connector transformation generation tool. The time required by X for this task is not affected by the number of call sites and the width of the interface, but may be affected (generally linearly) by the number of transformations that are composed.

Best case, my approach: Someone has performed the same modification, for the same connector type, on another system, and the code fragments are available. X does not need to think about how to compose it from transformations. X is able to reuse the existing code fragments and drop-in code. X only needs to change the system-specific parameters (the names of files to be read and modified). If X is *familiar* with the tool, it will take a few minutes per transformation to revise the parameters. If X is *unfamiliar* with the tool parameters, this task will take somewhat longer.

Average case, my approach: A similar modification is documented, here or in a cookbook of connector transformations. X does not need to think about how to compose it from transformations. The code fragments given in the cookbook may need to be modified slightly. X has to write a different set of drop-in code. (For example, X wants to perform a data translation like the one used in the VisAD case study, *but* with a different translation function; he must provide an implementation of that function.) In addition to the time spent in the best-case scenario, modifying the code fragments will take a few minutes per transformation if X is familiar with the approach, but may take longer initially if X is unfamiliar with the approach. Writing drop-in code will depend on X's familiarity with the domain: as in the traditional approach, X may only have to use a set of familiar library calls (about an hour), or X may have to work with an unfamiliar library (about a day). As in the best-case scenario, X also needs to provide system-specific tool parameters.

Worst case, my approach: (I assume that, at least, tool support does exist for X's connector.) X has to determine how to compose this domain-specific modification from generic transformations. For an experienced user, this will take an hour to several hours, depending on the complexity of the modification (in addition, if X is attempting to balance conflicting domain-specific considerations, such as dependability and performance, or dependability and security, X is likely to need more time to think about it). X must write code fragments for the generic transformations. For an experienced user, writing the code fragments for the transformations used in the case studies took about an hour per transformation. As with the average-case scenario, X also must write a new set of drop-in code: in the worst case, X will have to work with an unfamiliar library (about a day) or write an implementation of, for example, an error correction code himself (perhaps several days).

The final task is *deployment*. Having created a modified implementation, X must deploy it, test it, and fix any bugs that were introduced by the modification. This task is similar for either approach. In my approach there tend to be fewer *distinct* locations to introduce programmer error (I observed this tendency anecdotally in the VisAD case study and in the performance experiment that follows), but one mistake in the tool inputs will propagate to multiple locations in the tool outputs (once the initial mistake is corrected, the tool may be run again to regenerate the corrected outputs).

7.5 Performance of Generated Code

In this experiment I measured the communication latency of three versions of a simple benchmark system that has one RMI connector. In the first version, System A, the connector has not been modified at all. In the other two versions (B and C), I modified the connector to add the same collection of dependability enhancements described in section 7.2: retry, failover, duplicate detection, and logging/playback. In the second version, System B, I hand-modified the source code of the original system A; I hand-coded implementations of these dependability enhancements and placed them inline in the client and server. In the third version, System C, I used the tool and the connector transformations to automatically generate an augmented connector implementation from the base connector in system A.

System A represents a bound on the best possible performance for a modified connector (either hand-modified or generated), since this set of domain-specific enhancements is not intended to improve connector performance. By comparing the latency of calls in A and B, one can estimate how much overhead is intrinsic to the dependability enhancements themselves. By comparing B to C, one can estimate how much overhead is introduced by the use of connector transformations with unoptimized, automatically generated implementations.

7.5.1 Expectations

The latency of a remote method call in RMI (in system A) is composed of two parts: first, the time T_s spent by the stubs at either end in setting up and handling the call, and second, the network transmission time T_n. The second part (T_n) will vary depending on characteristics of the network: it will be smallest when the client and server are on the same host machine (I will attribute all latency in this case to T_s), and larger when the client and server communicate via a local area network (LAN) or wide area network (WAN). The latency of a remote method call in the modified RMI connectors (in systems B and C) includes a third part: the latency introduced by the modification, T_m.

A pleasing outcome for this experiment would be to find that the additional overhead in C is roughly comparable to the overhead T_s that is already present in the base connector; that is, one would be content if the latency for a same-host call in C is two or three times the latency for a same-host call in A or B. For a widely distributed application, even an order of magnitude increase ($T_m \leq 10 \times T_s$) might be acceptable since T_n on a WAN will dwarf it[7].

7.5.2 Details of the Experimental Setup

In system A, the server presents a remote interface with a single method that sends and returns an array of integers (`int[]`). In each *run*, the client makes a series of *trials*. In each trial, the client records the value of `System.currentTimeMillis()`, then makes 10,000 calls to the server, and finally calls `System.currentTimeMillis()` again and reports the elapsed time for the 10,000 calls. Within a run, the size of the array that is being passed back and forth is held constant. At the end of a run, the client reports the *average* of the elapsed times of 10 trials[8].

In all experiments, the client and server (and the backup server in systems B and C) were run on the *same host*, each in a separate Java Virtual Machine (JVMs). The host machine runs RedHat Linux on a 1.4 GHz AMD processor. The client and

[7] For example, an average "ping" round-trip time between two hosts at CMU, `gs235.sp.cs.cmu.edu` and `ux1.sp.cs.cmu.edu`, is 0.5 ms at the time of writing; average ping time from `gs235` to a colleague's machine on a campus network elsewhere in the U.S. (`maria.ccrc.wustl.edu`) is 44 ms.

[8] The first trial is discarded to allow the JVM to "warm up." The elapsed times of warm-up trials are reported but are not included in the final average.

Length of transmitted array	System A (unmodified)	System B (hand-modified)	System C **(generated)**
int[1]	6343 (100%)	6950 (109.6%)	14973 (236.0%)
int[10]	6405 (100%)	6803 (106.2%)	14925 (233.0%)
int[50]	6528 (100%)	6845 (104.9%)	14969 (229.3%)
int[100]	6636 (100%)	7079 (106.6%)	15537 (234.1%)

Table 7.2: Latency: milliseconds / 10,000 calls

server were compiled with Java 2 SDK 1.4.1 and used the corresponding version of Sun's JVM. Sun's RMI registry (`rmiregistry`) and a simple class server were also running on the same host. The server's class files were not visible to the client or the RMI registry and were downloaded by the client from the class server before each run. The registry's process and the class server's process were killed and restarted between A and B, and between B and C, to prevent any possible confusion in the registry between class files from different systems. For systems B and C, the server process and the backup server process were killed and restarted between each run: the reason for this is that the modifications to the connector include adding logging of call arguments at the client and caching of call results at the server; this ensures that both are empty at the start of a run.

7.5.3 Results

Table 7.2 shows the measurement results for systems A, B, and C. Each number in the table is an average over 10 trials. For system A, the latency is .6 milliseconds per call. For system B, the latency is .7 ms/call; the additional overhead (compared to A) can be attributed to the set of hand-optimized dependability enhancements. For system C, the latency is 1.5 ms/call: roughly 2.3 times the latency of A, or 2.2 times the latency of B. This result is within the range previously suggested as acceptable. (Recall also that the absolute overhead will appear smaller in comparison to the total latency for systems that are not based on a single host, due to network transmission time.) I discuss possible optimizations to the connector transformation implementation generator tool in 9.3.3, for single-host systems in which performance is a serious concern and communication is extremely frequent.

7.6 Usability Experience Report

The goal of this informal experiment is to determine whether a typical system architect who needs to create a new connector type can learn to use the connector transformation approach, including using the tool described in Chapter 6 to generate the modified connector implementation for his software system.

I asked a volunteer who was not previously familiar with the approach to perform a domain-specific connector modification using connector transformations. This volunteer represents a "typical" software engineer who is familiar with Java programming, has previously used RMI (in particular, knows how to compile and deploy a system that uses RMI), and is modifying a software system that is not unfamiliar to him. The implementation task in the experiment represents an "average case" (as described in section 7.4) scenario; that is, a case in which the user has access to "cookbook" examples of code fragments that instantiate the same *generic* connector transformations that the user needs, but that will require modification and/or different drop-in helper classes to provide the particular domain-specific modification necessary for the user's system. (For example, the examples might include an add-redirect transformation that implements a server-side *cache* of recent requests, a transformation used in the VisAD case study, while the user may actually need a add-redirect transformation that implements an *acceptance test*.) In addition if the user is able to recognize whether the selected transformations are commutative (or not commutative) and arrive at a correct ordering for the chain of tool-generated proxy implementations, this lends some support to the assertion in section 7.2.2 that correct ordering is not difficult for a typical user.

The user provided a small software system, with one client and one server, to be modified. Using this system as an example, I demonstrated the application of a single connector transformation and (to the extent of Chapter 6) explained what a connector transformation is, what kind of inputs the tool requires, and what kind of outputs the tool generates. I provided the user with a set of example code fragments and asked the user to add another specific simple modification to the connector to communicate an additional data field (this would require a "data translation" transformation at the client and the server). I remained at hand to assist with queries regarding tool use. The user was able to create the code fragments (by identifying and altering example fragments that instantiated a different domain-specific data translation) for a simple modification that appends a serial number to all client requests and prints and removes the serial number at the server. The user determined the ordering for the composition of the new and old transformations, provided the tool inputs, ran the tool to generate the proxy implementations, and compiled and tested the modified software system.

The experiment took approximately 2 hours: demonstration and exposition of the technique took an hour and 15 minutes, and the volunteer spent about 15 minutes selecting his transformation and modifying the code fragments, 20 minutes revising parameters to the tool, running the tool, and re-running the tool with corrected parameters (he gave the tool filenames in place of interface names, which resulted in compilation errors for the generated code), and about 15 minutes diagnosing typical RMI deployment mistakes (the user ran the "rmiregistry" process with the wrong CLASSPATH; then the user ran an older version of the server and his modified version of the client). After the user fixed the deployment problems, his system ran correctly.

The volunteer's qualitative feedback was that this version of the tool needs user interface improvement to make it easier to run, and that the tool parameters need to be better documented to avoid confusion between file names, interface names, and class names; but that the connector transformation approach seemed conceptually straightforward and that he felt able to perform similar ("average case") modifications in the future.

7.7 Formal Prediction

The goal of this experiment is to determine whether a reasonable linkage appears to exist between the implementations generated by the connector transformation tool and the formal interpretations given in Chapter 5. That is, do differences in formal properties, such as a prediction that two transformations are noncommutative, truly correspond to observable differences in implementation? If one discovers that two transformations are predicted to be noncommutative, yet there is *no* discernable difference when they are reordered in implementation, this would suggest a serious failure in the usefulness of the formalism. Conversely, although confirming that the predicted difference is discernable in the implementation proves nothing (rather, it fails to disprove something), it would provide a "sanity check" in the form of anecdotal evidence of a useful correspondence between the formalism and implementation.

In section 5.3.4 I described how to determine (through the use of safety properties, or, in some cases, by inspection of state machines) that the formal representations of two transformations are not commutative. In particular, I stated that the outcome of a failover transformation and a retry transformation, of the kind seen in the VisAD case study, is dependent on their ordering. But do the *implementations* of these transformations also turn out to be noncommutative?

To check noncommutativity of the implementation experimentally, I swapped the order of these transformations when applied to a Java RMI connector. (For demonstrative purposes, it is also desirable to add some test harness code to the server to cause it to return a timeout error whenever the user desires.) As it turns out, the difference in behavior is readily apparent by inspecting the amount of traffic received at the primary and backup servers. When the order is swapped, even a single error will cause the failover transformation to "fire", the primary server will cease to receive requests (even though it is still fully operational), and the backup server will show activity instead. When the order is correct (not swapped), a single error will *not* cause the failover transformation to fire, but killing the primary server (or using the test harness to cause multiple errors in a row so that the retry transformation "gives up") does cause it to fail over. (The preferred ordering is the one which is less likely to prematurely switch to the backup server.)

7.8 Conclusion

By composing and applying transformations to generate new enhanced connectors, it is possible to arrive at a *wide variety* of *useful* complex connectors.

The case studies described in this chapter show that it is possible to apply connector transformations to "real" connectors that are in use today in existing systems. Furthermore they show that it is possible to apply a range of domain-specific enhancements, and that multiple enhancements can be easily combined, if a little thought is taken to arrive at a correct ordering of interacting enhancements. The increase in latency introduced by automatically generated (and unoptimized) implementations of these enhanced connectors does not appear to be prohibitive. Results from the case studies suggest that the connector transformation approach offers time savings in the initial implementation, as well as the later modification, of connector enhancements; time savings are greatest when the person performing the enhancements is familiar with the connector transformation approach, and also vary with the level of reusable artifacts that exist for the specific desired enhancement (e.g., a similar domain-specific instantiation of a generic transformation, versus a novel instantiation of an existing generic transformation).

Chapter 8

Discussion

In this chapter I evaluate the major decisions made in the course of this research and their outcomes. I also list a set of difficult problems, related to creation of new connector types, that this work does not address.

8.1 Finding a Balance

A recurring theme in chapters 3 and 5 is the need to strike a balance between power and expressiveness, on one hand, and opposing considerations such as generality, on the other hand.

8.1.1 Transformation Granularity

The granularity of the transformations that I have selected was chosen according to two criteria.

First, I wanted to ensure that it would be possible to achieve *some* interesting results by applying only one or two transformations. The reasoning behind this is threefold. First, someone who is trying out the approach can then "do something" without having to immediately address concerns related to compositionality. Second, large compositions (regardless of the granularity of each transformation) seemed likely to result in implementation inefficiencies and higher bookkeeping overhead. Third, large compositions also seemed likely to increase the potential for adverse interactions (though it might also be the case that interactions between smaller transformations are easier to understand than interactions between larger ones, this seemed unlikely to compensate entirely for an n^2 growth of *potential* interactions).

My second goal, in opposition to the wish to limit the number of transformations in a "typical" composition, was that the set of transformations should be reasonably small (yet still provide a decent range of results). These are the reasons for keeping the set small: First, so that someone who is learning about the approach, or is considering which transformations to select for a particular domain-specific modification, is not overwhelmed with a long list; for an instance of a manageably-sized set, we might look to Gamma et al.'s Design Patterns reference book [19]. Second, so that one can reasonably expect that (for any connector type for which transformation generation tool support has been implemented), most or all of the meaningful transformations in the set will be supported. If there are a large number of transformations, it will be "too much work" for someone extending the tool to another base connector type.

It is easy to argue that these two goals have been achieved: the set of generic transformations is small, and some simple forms of the common dependability techniques discussed in Chapter 3 are achievable with one transformation.

However, this leaves unanswered the question of whether these were the right goals to achieve. What might the benefits be of selecting a larger or smaller granularity?

With a smaller granularity, each transformation is simpler and more restricted in effect; as a result, it could be possible formally to say more about a particular transformation or to provide interesting proofs. (This might even mitigate the effect of requiring a connector designer to think about ordering considerations of larger compositions of transformations.) However, without optimization, compositions of several transformations do result in measurable performance overhead (as seen in Chapter 7) incurred in passing through the "chain" of transformations, which would have to be considered.

With a larger granularity, the same kind of modification will involve a smaller composition, which will probably require less thought to correctly *order*, though each transformation may require more thought to *select* from a now-larger set. However, it may sometimes place connector designers in the position of a drugstore customer who is searching for a decongestant but is confronted only with "multi-symptom" cold medications: forced to select a transformation that does more, and is thus more costly, than is really necessary for the situation.

On the whole, the granularity selected may or may not have been the *optimal* granularity, but it did produce *useful* transformations. I would consider the outcome of this decision reasonable.

8.1.2 Balance in Formalism

When selecting the formalism that I used to express partial semantics for connector transformations (described in Chapter 5), there was a definite tradeoff between power and expressiveness, on one hand, and the complexity of the notation, on the other hand. Here I discuss the effects of this tradeoff in greater detail.

A simpler notation results in less expressiveness. FSP, for example, gives up the distinction between internal and external choice; this removes a major potential source of confusion and error for inexperienced users of formalisms, who now can't possibly use the "wrong" choice operator, but a "power user" will run into situations in which it is desirable to indicate which process is responsible for making a choice. For some notation simplifications, such as internal choice, it's still possible to express the idea in the simplified notation, but it requires cumbersome artificial conventions (which in turn appear somewhat confusing to inexperienced users) or several additional lines of simple notation (which result in a specification that is larger and more intimidating-looking, though with a smaller variety of actual symbols).

Is an FSP-based notation a reasonable compromise between simpler notation and expressiveness? I argue that, taking the humorous definition of a compromise as "something that makes all parties equally unhappy," it is. In the process of attempting to describe connector transformations, I have encountered occasional situations in which I noticed or regretted a lack of expressiveness. On the other hand, when presenting the simplest of the transformations to a general audience of computer science researchers (to say nothing of "ordinary" programmers), it is trivially apparent that the notation is still too hard to understand without substantial effort: an effort that, formal methods supporters argue, is ultimately amply rewarded, but nevertheless an effort that it is still difficult to persuade people to make.

8.2 Why Protocols?

When choosing an aspect of connector transformations to formalize, I selected protocols for several reasons. First, there are existing notations that appeared to have enough expressiveness and power that it was possible to do interesting things in them. Second, some of the analyses that these notations supported seemed likely to be of use to my target audience, software engineers; for example, if a proposed connector is going to deadlock, everyone likes to know it. Third, the notion of a protocol is likely to be somewhat familiar to this audience, and protocols provide a high-level view of a kind that they are likely to find understandable. For essentially the same reasons, I selected a protocol-based formalism that focuses on sequences of events (rather than other aspects of protocols, such as modeling trust or event timing); analyses based on event sequences catch kinds of problems that the target audience would care about and present corresponding event traces, which are conceptually not difficult to understand.

Protocol-based analyses are not an unreasonable choice for connector transformations, generally speaking. However, they are, in some respects, not the best possible match for the specific domain in which I have been exploring connector transformation applications, that is, *dependability*. When one wishes to say that an event is certain or is impossible, that a state is attainable or cannot be reached, the state machines and analyses offered by FSP are inviting. It is possible to determine whether (or not) the application of a particular transformation has made impossible the occurrence of a particular kind of event that represents some distinct class of errors. However, when attempting to model more realistic situations in which even the last backup server can fail, one would prefer to ask whether a transformation makes errors *less likely* to occur. Protocol-based formalisms such as FSP are not designed to address this question. In this respect, the two independent decisions to provide, on the one hand, an FSP-based notation, and on the other hand, concrete examples that are based in the domain of dependability, misses the opportunity to more easily provide analyses whose relevance to the examples is direct and inarguable. On the other hand, other domain-specific concerns, such as security or quality-of-service, similarly might be better addressed by yet a different formal model (e.g. [63]) and would gain nothing from an analysis of reliability; the utility of a protocol-based formalism is, at least, somewhat domain-independent.

8.3 Why Dependability?

I have just mentioned the decision to provide examples specifically in the domain of dependability. In order to keep the work manageable, I chose to restrict the investigation of domain-specific modifications to one particular domain. The alternative, to provide examples from multiple domains, with fewer examples from any particular domain, risks one way of failing to represent breadth of coverage of connector transformations. If, for example, I implemented on-the-fly compression for performance, and forward error correction for dependability, and private-key encryption for security, firstly, these would actually all be instantiations of *the same* connector transformation and thus would fail to exercise a breadth of the generic transformations, and secondly, this selection ought to provoke one to wonder whether I deliberately selected the easiest possible modification to represent each domain (the answer, for that triad of examples, would be "yes"). By instead restricting examples to a single domain, coverage of generic transformations and within-domain coverage will both be greater. (This restriction also sidesteps the possible investigation of inherent conflict, in intent or in side effects, between modifications from different domains; consider the results of applying the aforementioned triad simultaneously to a single connector, and it begins to look more interesting in a different sense. This issue will be revisited in section 8.7.)

Two questions arise. First, assuming the need to select one domain, was dependability a reasonable and representative choice? Second, can one make any extrapolations regarding applicability of connector transformations to other domains?

From the perspective, at least, of offering the opportunity to explore cooperating compositions of transformations, dependability turned out to be a reasonable choice. As seen in chapters 3 and 7, naive forms of some dependability modifications can be implemented with a single transformation, but more interesting and useful forms require multiple transformations that interact in some way. For example, one transformation may add data to the communication (such as a unique id) that is picked up and used by another transformation; or, one transformation may have to tell another transformation (such as the log playback) when to act. "Bad" interactions between the transformations that I used are possible but chiefly occur in the case of an incorrect ordering of transformations (for example, as has been noted, care must be taken when combining a data translation and an added role to ensure that the role's component does not receive data in an indecipherable intermediate format); perhaps another domain would have had more potential for interestingly unavoidable conflict. Dependability also offered a set of well-understood enhancements that have been used in the past (from which one might hang a claim that connector transformations that support these enhancements are "useful"); connector modifications are relevant to many of these, and, happily, the set of enhancements turned out to exercise a reasonable range of generic transformations. I have described in Chapter 3 how one would go about selecting transformations to implement a set of such enhancements, and I have demonstrated several in the case studies. On the whole, I believe that dependability was a reasonable choice both from the standpoint of potential utility of the results and of opportunity to exhibit the connector transformation approach.

Regarding the extrapolation to other domains, naturally applicability within dependability does not inherently guarantee the same degree of applicability within other domains. However, one can make an appeal to the similarities of structure between pairs of enhancement types from different domains; these similarities originally motivated the formulation of "generic" connector transformations. Connector transformations that are useful in dependability, for translating data to a different (more redundant) format or for adding a component that rejects some communications (that fail an acceptance test), might be expected to be useful in, for example, security, for translating data to a different (more cryptic) format or for adding a component that rejects some communications (that fail a check of authorization or access privileges). More compellingly, in earlier work [56] (summarized in section 7.1) I have argued, by composing several connector transformations to add Kerberos authentication to a Java RMI connector, that my approach can also be usefully applied in the area of *security*.

8.4 Reflections on Formal Insights

In ordering the work performed for this thesis, I chose to implement some transformations before the investigation of formalisms for describing connector transformations, and I implemented other transformations after the formal templates had been established. As it happens, this alternation provided an opportunity to compare whether there were insights gained from the formalism that make it easier to construct implementation tools. Anecdotally, my experience was that it *was* helpful, during the implementation, to have a less-informal characterization of what a transformation "needs" as inputs, what parameters should exist and be possible to vary, and how a transformation to be implemented is structurally similar to or different from existing transformations. Without this level of advance thought, earlier efforts in implementing specific transformations tended to suffer from gradual "re-thinking" as I discovered either inadequacies (such as the need to be able to perform actions "after" a call, not only in the case when the call returns a value but also in the case of calls that do not return a value, which the formal characterization would have made immediately apparent) or, conversely, non-necessities (offering more flexibility than is commonly needed resulted in proliferation of "required" tool inputs and, ultimately, user intimidation) in their implementation.

8.5 Extending the Scope

Suppose that one wished to extend this work to another domain (besides dependability) or another category of connectors (besides communication-based).

8.5.1 Principles of the Approach

When I describe the approach as "principled" as compared to an ad-hoc approach, what are the principles? These principles will provide guidance for anyone seeking to extend support to a different domain.

First, in the connector transformation approach, a large modification is divided into smaller, simpler, composed modifications. This can be thought of as a hybrid of divide-and-conquer and an orthogonal basis set with composition.

Second, this approach constrains possibilities by providing a finite set of generic transformations that can be varied in certain ways. The principle at work here is discipline with appropriate liberty: variation should be possible but not wild variation.

Third, the tool-based generation takes a single architecture-level idea (a generic transformation), instantiates it with user-provided parameter values and code fragments, and performs necessary changes and insertions across the multiple affected concrete artifacts that make up the connector implementation. That is, this principle is to reify and localize.

8.5.2 Desirable Characteristics of an Enhancement Domain

What characteristics of the dependability domain made it a suitable target for connector transformations and should be sought in other candidate domains?

First, the space of enhancements in the domain should already be understood. I used Lyu's [35] survey of well-known fault tolerance techniques. Such a survey for another domain would be desirable as a starting point.

Second, one would wish to see enhancements that are independent of one another or that are cooperative (and reasonably limited) in their interaction. For example, in dependability one might expect to use some enhancements together cooperatively (e.g. one supplies some specific information or a tentative diagnosis to another) or without adversely affecting one another. An example of a domain in which this characteristic is unlikely to hold is security in which (though an individual enhancement might be constructed from connector transformations and used) one would have to use extreme caution when combining multiple enhancements in order not to introduce emergent properties such as security holes.

Third, this approach is local in nature (enhancements are applied to specific connector instances) and may not be suited for a domain in which modifications require global knowledge. Also, naturally, since the approach deals with connectors, the enhancements of the domain should be related to interaction and not wholly intra-component. For example, some performance enhancement techniques are unlikely to be appropriate candidates for these reasons (though techniques that involve component replication may actually be suitable).

8.5.3 Reflections on Non-Communication Connectors

In this work I deal only with communication-based connectors (interaction mechanisms that enable software components to *communicate* with one another). How much of the approach of software transformations could be carried over to *other* kinds of connectors? As an example I'll consider a real-time-scheduling connector.

Such a connector can be augmented via transformations. Are they the same set of transformations? No. There could be some overlap in that some of my communication-connector transformations have an equivalent that would be applicable, but some do not. Consider "add a new role to the connector;" where the existing transformations deal with how the new role will observe and participate in communication, equivalent transformations would deal with how the new role can be made to fit into an existing scheduling policy. (Is it a first-class participant, scheduled on the same basis as the others, similar in spirit to an "add parallel" transformation where the new role is active whenever the existing role it replicates is? Or perhaps does it share the scheduled time of some existing role, similar in spirit to an "add switch" transformation where the new role and the old role it replicates are essentially taking turns?) An example of a transformation that would probably not have an equivalent is the sessionize transformation; it is common to have "sessions" in a communication (time intervals during which some piece of state is saved and reused) but not in scheduling.

More specific parts of this research include the formal support and implementation generation approach. Since the formalism is based on protocols it is necessarily biased toward the description and properties of connectors that use communication protocols. A different formalism would be more appropriate and more natural for the concerns of a scheduling-based connector. As for the tool approach, which makes use of "hook-in points" as an abstraction of locations within code or noncode artifacts that are (at a concrete level) specific to a connector implementation, since non-communication connectors too have the same issues of being spread across multiple implementation artifacts, such an approach would still be needed and part of the tool implementation itself could still be of use. There would also be some overlap in the specific set of hook-in points (some have a commmunication bias, e.g. "before call", but others, e.g. "on initialization", are communication-independent).

8.6 Implementation Decisions

Two decisions made in the creation of the tool, to use a proxy-based approach and to assume availability of source code, were made in part to facilitate the implementation of preliminary versions of the tool. However, these decisions have an impact on the kinds of base connectors and systems that it is easiest for the tool to support. Here I describe the limitations imposed on this specific tool implementation by these decisions.

8.6.1 Assumptions about Base Connectors

The implementation approach described in Chapter 6 leverages the lookup services and proxy generation mechanisms of the basic connector type being modified. These provide a straightforward means of inserting proxies in connectors that have these attributes. The tool can recognize the name of a lookup call, which is specific to that connector type, and replace its return value with a newly created proxy; in the receiving component, the tool can similarly recognize and replace a call that registers that component with the lookup service.

Some connector types, however, do not offer these opportunities. They may instead call a communication library that has a well-defined API; one can in that case recognize relevant API calls and replace them (or perhaps provide a replacement for the library itself). These modifications are more intrusive, as there will probably be more than one call to send traffic (as compared to one lookup call in the previous case), but should still be effective for transformation of outgoing traffic. For incoming traffic, provided that the API includes registration by components that wish to receive traffic, one can restrict modifications to a smaller number (the number of registration calls). Thus, tool support can be extended to some kinds of connectors that do not meet the assumptions described in Chapter 6 without entirely revising the implementation approach or the kinds of inputs required of the tool user, but increased work is required for the person implementing the tool support.

8.6.2 Modification of Source Code

The implementation approach taken in this work assumes that the source code for the *application* is available and can be modified. For many situations this is a reasonable assumption: a development team is assembling an application from components that were previously or recently written, and has the source code.

In other cases, however, such as an "end-user" programmer who has purchased a set of shrink-wrapped applications and is assembling them into a larger cooperating application with the help of some glue code, source code will be *unavailable* for some or all of the "components." These situations, furthermore, are ones in which connection-related issues such as mismatch are likely to arise. To the extent that connector transformations can help to resolve these issues, one would like to be able to use them.

I make use of source code modification to introduce the proxies that are generated by the connector transformation tool. Can these proxies be introduced in some other way? Certainly there are techniques that can be used to intercept calls on their way from the application level to lower-level code; one can, perhaps, intercept system calls or replace a dynamically linked library. However, in addition to modifying source code I also rely on the ability to read source code, or, more specifically, to read an *interface definition* so that a proxy that "looks like" that interface can be generated. If the APIs for the sourceless components are well documented, one might, with a little extra work, write a matching interface definition that can be used by the tool. If an API is not accurately documented or, in the worst case, is undocumented (perhaps because one wishes to affect the communication between two components that the component provider did not think of as "user-serviceable parts"), it will be difficult to use this tool to generate proxy code.

8.7 Unsolved Mysteries

This work addresses the problem of constructing new connector types. It ameliorates some aspects of this problem, such as the need to make many non-localized changes and the difficulty of maintaining interwoven changes in an ad-hoc approach. There are some difficulties related to the construction of new connectors, however, that are out of the scope of this work.

I provide domain-specific instances of connector transformations that address dependability concerns. When constructing a software system, one often has concerns in multiple extrafunctional domains. Some of these concerns are *inherently* in conflict. For example, some dependability techniques involve replication of data and services to decrease the chance that *all* instances will fail; from a security perspective, however, replicating a service *increases* the chance that at least one of the instances will be compromised. Other concerns may be attainable together, but only by taking far greater care in the design of the modification than would be necessary for either concern alone (for example, attempting to increase both dependability and performance). Determining appropriate tradeoffs between desired quality attribute measures (as in the Architecture Analysis Tradeoff Method [30]) and resolving conflicting extrafunctional properties are difficult engineering issues that requires human attention and that this thesis does not address. At best, the connector transformation approach may make subtle conflicts more apparent than they would be in an ad-hoc connector modification approach.

I identify "splice" as a category of connector transformation. However, the facilities I provide for splice transformations are extremely limited. Resolving forms of communication mismatch, such as protocol mismatch [64], is an interesting research problem in its own right, and one that this work does not address.

I identify "aggregate" as a category of connector transformation and further identify a need to restrict switching between protocols to "safe points." However, I do not provide a facility for determining, for an arbitrary connector type, what these safe points may be. This is a difficult research problem in its own right, which has been touched on (for some connector types) in Dynamic Wright [2]. Note that if a human *incorrectly* designates a point in the connector protocol as a safe location to switch to another protocol, problems may be obvious, such as deadlock, or may be more subtle and difficult to discover.

While the formalism in my approach can be used to discover compositions that are noncommutative, it does not give much assistance to the human in determining which ordering is preferable (in cases where neither ordering results in deadlock). One can attempt to phrase formal properties that a desired ordering should achieve (for example, end-to-endness of encryption) and run analyses to check these properties, but it is up to the human to know (and specify) what these goals are.

Chapter 9

Conclusions and Future Work

> *"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."*
>
> — *ALICE'S ADVENTURES IN WONDERLAND, Lewis Carroll*

9.1 Summary

In this dissertation, I have demonstrated that

> *We can define a small set of basic transformations that, when applied compositionally to simple communications-based connectors, produce a wide variety of useful complex connectors. These transformations can also be given a formal interpretation that allows us to understand their properties with respect to communication protocols. Furthermore, we can build a transformation-based tool that facilitates the generation of implementations of these new connectors.*

To demonstrate this claim, I identified such a set of transformations and developed a formalism and an implementation generation tool. In Chapter 3, I described the connector transformation approach which enables rapid construction of new variants of connector types, and introduced in Chapter 4 a set of non-domain-specific connector transformations which can be used to produce a range of domain-specific enhancements to a basic connector type.

In Chapter 5, I demonstrated that formal notation can be used to express connector transformations in a way that enables useful analyses (such as transparency with respect to interfaces, freedom from deadlocks, and ability to make progress). Using an approach that describes connectors as structured protocols, I described connector transformations as transformations on these protocols. I showed how to check that two transformations are composable but noncommutative and how to check that a dependability-enhancing transformation eliminates a class of error events observed at the interface of an unreliable connector. This chapter validates the claim that connector transformations can be given a formal interpretation that supports understanding of their properties.

I created a tool, described in Chapter 6, that supports the semi-automatic generation of implementations of new complex connector types using connector transformations and basic connector types. To validate that this tool can be used on multiple connector types to add multiple commonly-desired enhancements, I conducted two case studies described in Chapter 7. In each of these case studies, I applied a combination of at least three connector transformations to connectors in existing software systems of moderate size. Together these transformations represented five kinds of common dependability-related enhancements. I found it easy and fast to make small policy changes to enhancements. Using these case studies and a novice user's experience report, I provided an estimated breakdown of the tasks required in the connector transformation approach as compared to an ad-hoc modification approach; the times experienced in the connector transformation approach indicate that this approach is likely to save time. I also compared the same-host latency for an unoptimized generated connector implementation (incorporating several connector transformations) to a hand-written equivalent and showed that the increase in latency is not unreasonable. This chapter supports the claim that connector transformations can produce a *wide variety* of *useful* connectors.

Chapter 8 presents a critical evaluation of the connector transformation approach and related issues. Finally, in the current chapter, I discuss the contributions of the thesis and potential opportunities for future work that will build on the connector transformation approach.

9.2 Contributions

This work's contributions to the field of computer science fall into four areas.

A new approach for constructing modifications to component interactions. This thesis describes an approach of separating a monolithic modification into a set of smaller composable connector transformations. Independent of the specific formalism and prototype tool implementation produced in this thesis, this technique benefits developers of new connector variants by reducing development costs and by producing a more understandable and maintainable modification. It benefits designers of component-based systems by widening the field of available connector types.

A new application of generation technology. This thesis describes an approach in which a diverse range of new connector implementations can be semi-automatically generated from a small set of base connectors by applying domain-specific instantiations of connector transformations. It benefits connector developers by providing a basis for rapidly generating new connector variants. It benefits software tool builders by providing a technique for building generators of connector variants. The technique may also be generalizable to variant-generation of other kinds of specialized artifacts besides software connectors.

A formalism that is targeted at software engineers. The approach focuses on performing analyses that are intended to address engineering concerns and on providing a system of parameterized templates that help to specialize an existing formal notation of reasonable complexity so that it can be used to describe an aspect of connectors and connector transformations. In addition to benefiting the target audience, it benefits researchers in formal methods by providing an example of the idea of tuning formalism to a particular target audience.

An approach to formalism that incorporates reusable parameterized fragments. In order to save some effort for people who have a sufficiently firm grasp of formalism to be able to understand and apply existing descriptions, Chapter 5 provides parameterized templates for connector transformations. In addition to benefiting the target audience, this approach benefits researchers in formal methods by demonstrating a possible means of providing greater reusability in the construction of formal descriptions.

A new view into the space of software connectors. By considering complex connectors in terms of compositions of connector transformations and simpler connectors, the space of connectors appears simpler and more orderly. It benefits researchers in the field of software architecture by providing a different means of categorizing existing connectors. It benefits connector developers by revealing hidden similarities between variants of connector types, which may lead to reuse and generalization of designs, and by directing attention to underpopulated regions of the space.

9.3 Future Work

The connector transformation approach offers opportunities both in the short term for investigating extensions and improvements to the artifacts contributed by this work and in the longer term for interesting research directions.

9.3.1 Short-Term

Several directions for building in a small way on the results of this work suggest themselves. These include:

- Comprehensive user studies. In order to better support the process of connector development, it is necessary to determine where the "difficult" parts of the connector transformation approach lie; this will suggest tasks for which more assistance should be provided.

- Increased breadth of implementation support, both in the number of basic connector types and in the collection of domain-specific recipes (currently limited to a representative set of dependability enhancements).

- Incorporating and building on existing techniques for generating "component adaptors" to provide a sound basis for the creation, application, and composition of "splice" transformations, given specifications of mismatched components.

- Incorporating and building on additional kinds of formalism and analysis techniques, to provide analyses of connector transformation properties for which protocol descriptions are awkward, inadequate, or irrelevant.

- Improvements in code generation to automatically merge portions of the proxy chain that are generated for compositions of transformations, to optimize out some of the latency introduced by the layers of generated code.

- Categorization of the kinds of optimization that are possible if access to the source code for the middleware is available.
- Integration of the formal analysis and the implementation generation support into an architectural design environment such as AcmeStudio, including assembling a "palette" of connector transformations and providing a means of specializing, composing, and applying them to a software system.

However, longer term possibilities are of greater interest.

9.3.2 Dynamism!

The generational approach outlined in this thesis is *static*. Connectors are modified in advance of their use; only pre-defined changes in behavior, such as switching between a set of protocols that has been determined in advance, are possible at run time. While this approach is limiting, it also ensures that an opportunity exists, between the modification and the use of a connector, for a human to validate properties of the connector.

A dynamic approach to connector modification, enabling connectors to be reconfigured or changed "on the fly" in ways that were not foreseen when the system was deployed, would support current research topics such as self-healing systems. A vital piece of this work would be to resolve or mitigate the safety considerations associated with allowing connectors to modify themselves in unforeseen ways.

9.3.3 Integration of Formalism

The correspondence between the formal description and the implementation of connector transformations is not enforced in the approach described here. Such a correspondence is desirable but would be nontrivial to achieve.

The implementation generation tool described in Chapter 6 could in principle help to enforce a simple correspondence between specific wrapper processes (in the formal expression of the transformation) and equivalent added and modified code artifacts (in the implementation). Providing a higher level of guarantee of process/artifact correspondence would aid in traceability of problems, from their discovery in one, to the equivalent in the other.

Another possible direction for this work is the actual semiautomated derivation of a formal description from the kind of "code fragments" used in the generation tool, and, in the other direction, deriving such code fragments from formal descriptions. Describing formal transformation templates in terms of their actions (redirect, record/replay, insert, replace, or discard) on events could enable automatic generation of instances of some kinds of specifications, given a connector specification and a set of inputs that include the template and the affected elements of the connector. This work would include determining what additional information is required from a human in order to make translation in one direction or the other possible. Investigating what (if anything) can be proven about correspondence of implementation transformations and formal transformations would be of interest.

9.3.4 Formal Methods for . . . Dummies?

FSP is anecdotally easier to use than other, more expressive process algebras, but certainly it still has a non-trivial learning curve. When presenting FSP-based formalism, I have usually been asked about the possibility of making it easier still. This research direction would require discovering what are the aspects of formal notation that people find difficult, and investigating alternative, perhaps more visual, approaches that still support an expandable collection of analyses that software engineers could find useful.

9.3.5 Generation Tools for Generation Tools

Currently, the process of extending generational tool support to additional connector types is time-consuming. This research direction would involve investigating ways of generalizing or streamlining the process of extending generational tool support to additional connector types, to provide support for *tool* generation. First steps would include more rigorously defining the needs of the tool (which I have described only informally in this document) and categorizing common implementation techniques that can be used to provide these needs, such as common techniques (including those that are operating-system specific and language specific) that may be used for interception of library calls.

9.4 Conclusion

In this dissertation I have illustrated a new approach to the construction of complex connector types. I have given a set of generic, composable transformations that can be applied to existing connector types to produce more complex connectors. This process of applying connector transformations has an equivalent in formal notation, where I give a means of producing a protocol-based formal description of the new connector type. I have demonstrated, through case studies, a prototype tool that generates implementations of enhanced connector types by applying domain-specific instantiations of connector transformations to an existing connector implementation.

Connectors in software architecture design provide a means of embodying interactions between components, a useful abstraction for the more complex and nonlocalized nature of their equivalent in a software system implementation. Similarly, connector transformations provide a more principled and abstract way of looking at complex enhancements to connectors. The approach, the tool support, and the techniques presented here do not resolve all issues that relate to the task of incorporating additional extrafunctional properties in a software system. However, by clearing away some of the tedious problems involved in reifying such modifications, connector transformations will ultimately help to liberate designer attention to focus on those issues that, by their nature, *require* human decision-making.

Bibliography

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

[2] Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998. An expanded version of the paper "Specifying Dynamism in Software Architectures," which appeared in the Proceedings of the Workshop on Foundations of Component-Based Software Engineering, September 1997.

[3] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[4] Robert Allen, David Garlan, and James Ivers. Formal modeling and analysis of the HLA component integration standard. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Buena Vista, Florida, November 1998. ACM.

[5] Farhad Arbab, Christel Baier, Jan Rutten, and Marjan Sirjani. Modeling component connectors in Reo by constraint automata. Technical report, Centrum voor Wiskunde en Informatica (CWI), 2003.

[6] Farhad Arbab and Jan Rutten. A coinductive calculus of component connectors. Technical report, Centrum voor Wiskunde en Informatica (CWI), 2002.

[7] A. Avizienis, J.-C. Laprie, and B. Randell. Fundamental concepts of dependability. Technical report, UCLA CSD Report no. 010028, 2001.

[8] A.A. Avizienis. The methodology of n-version programming. In Michael R. Lyu, editor, *Software Fault Tolerance*. John Wiley and Sons, 1995.

[9] Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Alexander E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, October 1996.

[10] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions of Software Engineering and Methodology*, pages 355–398, October 1992.

[11] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. *ACM Conference on Lisp and Functional Programming*, pages 55–64, June 1994.

[12] Grady Booch. *Software Components with Ada*. Benjamin/Cummings, 1987.

[13] 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*, pages 1–11. ACM Press, 1990.

[14] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[15] Shang-Wen Cheng, David Garlan, Bradley Schmerl, Joao Sousa, Bridget Spitznagel, Peter Steenkiste, and Ningning Hu. Software architecture-based adaptation for pervasive systems. In *International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing (ARCS'02)*, Karlsruhe, April 2002. Springer LNCS #229, pages 67-82.

[16] 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 21st International Conference on Software Engineering*, pages 3–12. IEEE Computer Society Press, 1999.

[17] Robert DeLine. *Resolving Packaging Mismatch*. PhD thesis, Carnegie Mellon, School of Computer Science, 1999. Issued as CMU Technical Report CMU-CS-99-141.

[18] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1995.

[20] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, April 1995.

[21] David Garlan, Robert T. Monroe, and David Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON'97*, Ontario, Canada, November 1997.

[22] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[23] William Hibbard, Curtis Rueden, Steve Emmerson, Tom Rink, David Glowacki, Tom Whittaker, Don Murray, David Fulker, and John Anderson. Java distributed objects for numerical visualization in VisAD. *Communications of the ACM*, 45(4ve):160–170, April 2002.

[24] Matti A. Hiltunen and Richard D. Schlichting. Constructing a configurable group RPC service. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS-15)*, May 1995.

[25] Dan Hirsch, Sebastian Uchitel, and Daniel Yankelevich. Towards a periodic table of connectors. In *Simposio en Tecnolog a de Software (SOST '99)*, 1999.

[26] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[27] Ningning Hu. Network aware data transmission with compression. In *Selected Papers from the Proceedings of the Fourth Student Symposium on Computer Systems (SOCS-4)*. Carnegie Mellon University School of Computer Science Technical Report, CMU-CS-01-164, October 2001.

[28] Hermann Hueni, Ralph E. Johnson, and Robert Engel. A framework for network protocol software. *Proceedings of OOPSLA'95*, pages 358–369, 1995.

[29] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, WA, July 1999.

[30] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, and Jeromy Carriere. The architecture analysis tradeoff method. In *The Fourth International Conference on Engineering of Complex Computer Systems (ICECCS98)*, Monterey, CA, August 1998.

[31] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[32] K.H. Kim. The distributed recovery block concept. In Michael R. Lyu, editor, *Software Fault Tolerance*. John Wiley and Sons, 1995.

[33] J-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Architectural issues in software fault tolerance. In Michael R. Lyu, editor, *Software Fault Tolerance*. John Wiley and Sons, 1995.

[34] Antnia Lopes, Michel Wermelinger, and Jos Luiz Fiadeiro. A compositional approach to connector construction. In *Recent Trends in Algebraic Development Techniques*, volume LNCS 2267, pages 201–220. Springer-Verlag, 2001.

[35] Michael R. Lyu, editor. *Software Fault Tolerance*. John Wiley and Sons, 1995.

[36] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 1999.

[37] Nenad Medvidovic, Marija Mikic-Rakic, and Nikunj Mehta. Improving dependability of component-based systems via multi-versioning connectors. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*. Lecture Notes in Computer Science (LCNS 2677), 2003.

[38] Nenad Medvidovic, Peyman Oreizy, and Richard N. Taylor. Reuse of off-the-shelf components in C2-style architectures. In *The 1997 Symposium on Software Reusability (SSR'97)*, pages 190–198, Boston, MA, May 1997. Also Proceedings

of the 1997 International Conference on Software Engineering (ICSE 19), Boston, MA, May 17-23, 1997. pages 692-700.

[39] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[40] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 295–304, Limerick, Ireland, June 2000.

[41] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[42] Priya Narasimhan, Louise. E. Moser, and P. Michael Melliar-Smith. Exploiting the Internet inter-ORB protocol interface to provide CORBA with fault tolerance. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS).* USENIX, 1997.

[43] B. Clifford Neuman and Theodore Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[44] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[45] J. Postel. Transmission control protocol. Technical report, RFC-793, 1981.

[46] B. Randell and J. Xu. The evolution of the recovery block concept. In Michael R. Lyu, editor, *Software Fault Tolerance.* John Wiley and Sons, 1995.

[47] L. Sha, J. Goodenough, and B. Pollack. Simplex architecture: Meeting the challenges of using COTS in high-reliability systems. *Crosstalk*, April 1998.

[48] Mary Shaw. Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[49] Mary Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. In *Proceedings of the Symposium on Software Reuse (SSR'95)*, April 1995.

[50] Mary Shaw and Paul Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *21st International Computer Software and Applications Conference (COMPSAC97)*, pages 6–13, Washington, D.C., August 1997.

[51] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and implementations for architectural connections. In *Third International Conference on Configurable Distributed Systems*, May 1996.

[52] Charles Shelton, Philip Koopman, and William Nace. A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems. In *Proceedings of the 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, January 2003.

[53] Reid Simmons and Dale James. Inter-Process Communication: a reference manual. http://www.cs.cmu.edu/afs/cs/project/TCA/ftp/ipc.ps.gz, 2001.

[54] Reid Simmons and Greg Whelan. Visualization tools for validating software of autonomous spacecraft. In *The Fourth International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS)*, Tokyo, Japan, August 1997.

[55] Gurdip Singh and Zhenyu Mao. Structured design of communication protocols. In *IEEE International Conference on Distributed Computing Systems*, May 1996.

[56] Bridget Spitznagel and David Garlan. A compositional approach for constructing connectors. In *The Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, pages 148–157, Royal Netherlands Academy of Arts and Sciences Amsterdam, The Netherlands, August 2001.

[57] F. Stomp and W. de Roever. Designing distributed algorithms by means of formal sequentially phased reasoning. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, 1989.

[58] Sun Microsystems. Java Message Service. http://java.sun.com/products/jms.

[59] Sun Microsystems. Java Remote Method Invocation (RMI). http://java.sun.com/products/jdk/rmi.

[60] Sun Microsystems. Java Pet Store Demo 1.3.1. http://java.sun.com/blueprints/code/jps131/docs/, 2002.

[61] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., and Jason E. Robbins. A component- and message-based architectural style for GUI software. In *Proceedings of the Seventeenth International Conference on Software Engineering (ICSE-17)*, pages 295–304, Seattle, Washington, April 1995.

[62] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. Technical report, Cornell/TR97-1638, 1997.

[63] Nalini Venkatasubramanian, Carolyn Talcott, and Gul Agha. A formal model for reasoning about adaptive QoS-enabled middleware. In *Proceedings of Formal Methods Europe (FME 2001)*, Berlin, Germany, April 2001.

[64] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, mar 1997.

[65] Matthew J. Zelesko and David R. Cheriton. Specializing object-oriented RPC for functionality and performance. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, Hong Kong, May 1996.