A Compositional Formalization of Connector Wrappers

Bridget Spitznagel
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
sprite@cs.cmu.edu

David Garlan
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
garlan@cs.cmu.edu

Abstract

Increasingly systems are composed of parts: software components, and the interaction mechanisms (connectors) that enable them to communicate. When assembling systems from independently developed and potentially mismatched parts, wrappers *may be used to overcome mismatch as well as to remedy extra-functional de£ciencies.*

Unfortunately the current practice of wrapper creation and use is ad hoc, resulting in artifacts that are often hard to reuse or compose, and whose impact is dif£cult to analyze. What is needed is a more principled basis for creating, understanding, and applying wrappers. Focusing on the class of connector wrappers *(wrappers that address issues related to communication and compatibility), we present a means of characterizing connector wrappers as protocol transformations, modularizing them, and reasoning about their properties. Examples are drawn from commonly practiced dependability enhancing techniques.*

1. Introduction

Increasingly systems are implemented as compositions of independently-developed components that must be integrated into working systems using various interaction mechanisms, such as remote procedure call, event buses, pipes, etc. For such systems a serious problem is dealing with component mismatches that arise when the expectations of a component do not match those of other components or of the environment into which it is placed [4].

For example, a component selected for use in some software system may not use the same units of measurement or the same data format as the rest of the system. Or, a COTS component may not gracefully tolerate out-of-range input data produced by other components.

In this setting the traditional approach of directly rewriting or modifying the software to solve the problem may not be feasible, since components are often built by third parties, or are suf£ciently complex that rewriting them is not cost-effective.

One widely used technique to deal with this problem is to use wrappers. Informally a *wrapper* is new code interposed between component interfaces and communication mechanisms. The intended effect is to moderate the behavior of the component in a way that is largely transparent to the component or the interaction mechanism.

For example, a unit conversion mismatch might be resolved by a data conversion wrapper that intercepts data entering or leaving the offending component, and converts it to the units expected by the rest of the system. Or, in the case of a component that does not gracefully handle out-of-range data, a wrapper might be constructed to harden the component by rejecting illegal input, allowing only well-tolerated inputs to reach the component, and thereby increasing the reliability of the system as a whole [15].

Unfortunately the current practice of wrapper creation and use is ad hoc and something of a black art. To overcome a (perhaps unforeseen) dif£culty quickly, a one-off wrapper is written speci£cally for that problem in that software system. Consequently as somewhat random pieces of code, wrappers are often hard to reuse elsewhere, analyze, compose with one another, modify, and maintain.

As a result, many important questions that might arise cannot be answered. For example: How does a speci£c wrapper, interposed between a component and a connector, affect the protocol of that connector? Are existing properties of the protocol maintained; does a desirable new property emerge? Do two potential wrapping efforts interact in bad ways? Does it matter in which order the wrappers are applied? Does a wrapper violate the interface expected by the component that is being wrapped? Does a wrapper require modi£cations to the source code of a component?

In principle, such questions could be answered in an ad hoc fashion themselves, for example, by writing a formal description of the affected part of the system as it would stand after the incorporation of the wrapper. However this task would have to be repeated each time another wrapper is added, and yields no general understanding. Moreover, the wrappers themselves would still not enjoy the usual desired properties of reusability, maintainability, and so on.

What is needed is a more principled basis for wrapper creation and application, ideally providing three desirable capabilities: First, we would like to be able to *specify* a wrapper itself, independent of any particular context of use. Second, we would like to use this speci£cation to understand things such as *impact* of its use, its effects on the communication protocol between components, compositional properties, etc. Third, we would like to be able to relate the wrapper speci£cation to an *implementation* using tools and techniques that help enforce that the implementation corresponds to the speci£cation.

In this paper we address the £rst two of these issues for an important subclass of wrappers, namely *connector wrappers*.[1] As we detail later, these wrappers are used to repair or augment communication-related properties of a system, such as the two examples above. As we will show, it is possible to specify this class of wrappers as modularized protocol transformations, whose properties can be reasoned about using standard notations and tools. We give examples drawn from commonly practiced dependability-enhancing techniques, show the application of these example wrappers to two connector speci£cations, demonstrate the ease of composition of this kind of wrapper speci£cation, and illustrate how analyses can be used to con£rm whether the wrappers actually achieve their intended purpose.

2. Related Work

The widespread use of wrappers (which, in part, motivates this work) has given rise to efforts directed at standardized support for wrapper insertion. System-level support mechanisms, usually called *interceptors*, have become increasingly 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 [13] and security [3] of COTS components, or to add instrumentation [8]. However, by their nature these efforts are speci£c to a connector implementation and/or set of system libraries; and, generally speaking, do not address the questions posed in section 1, which are concerned with what is going on in the wrapper itself rather than the means of wrapper emplacement.

Our work builds on process algebras such as CSP and CCS [6, 12]. In particular, it applies to FSP [11] some of the structure of Wright [1] 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 (roles) and interactions (glue), enables explicit identi£cation of the communicating parties and their obligations, as well as compatibility checks. The work described in this paper, however, goes beyond that of previously published work on Wright by further decomposing the connector and promoting reusability of "wrapper" interaction elements.

We also make use of the idea, drawn from software architecture, of treating a software connector as a separate £rst-class entity [18], on a par with software components. Building on the idea of £rst-class connectors, and with similar goals to our work, 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 CommUnity, a Unity-like parallel program design language [9]. The categorical semantic underpinnings of their work gives it a somewhat different form of compositional framework, as compared to the use of process algebras (as in our work).

Mismatch resolution, a problem that gives rise to some wrappers, has also been tackled in software architecture. When two mismatched components are unable to communicate via existing connectors, one alternative to wrappers is to construct or modify a *connector* to resolve the mismatch [19]. Another technique, Flexible Packaging [2], 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. The ¤exible packaging approach is elegant, but lack of adoption by component providers makes it unlikely to replace practitioners' use of wrappers in the immediate future.

Also related to this work is protocol synthesis, which deals with a protocol's composition from (or decomposition into) simpler protocols. Ensemble [21] enables the construction of an adaptive protocol composed of stacked micro-protocol modules. The x-Kernel [14] project has also used micro-protocol composition to design and implement a dynamic architecture for ¤exible protocols that take advantage of operating system support for ef£cient layering. Conduits+ [7] 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 ¤ow through the conduits). The work described here has a somewhat different goal: to describe the impact of application-level wrapper code on a pre-existing protocol, rather than to construct a fresh protocol (and corresponding connector implementation) from scratch.

[1]We address the third issue in [20].

3. Overview of Approach

Informally a *wrapper* is new code interposed between component interfaces and infrastructure support (e.g., between application-level code and communication facilities such as RPC). The intent of the code is to alter the behavior of the component with respect to the other components in the system, without actually modifying the component or the infrastructure itself.

An important class of wrappers are those that are primarily designed to affect the communication between components. We refer to these as *connector wrappers*. Connector wrappers encompass a wide range of behaviors, including things such as changing the way data is represented during communication, the protocols of interaction, the number of parties that participate in the interaction, and the kind of communication support that is offered for things like monitoring, error handling, security, and so on.

Moreover, since connector wrappers focus on modifying the behavior of shared communication infrastructure, they are not inherently speci£c to the particular components being wrapped. As a result, they have a greater potential for reuse and generalization. For example, a connector wrapper that adapts a communication to use encrypted data could be reused between many components.

3.1. Goals

Our goal is to provide a more formal, disciplined approach to connector wrapper design (and indirectly implementation), so that we can understand their behavior and other properties. Speci£cally, there are three important classes of properties that we would like to analyze:

- Soundness: Having introduced a wrapper (or sequence of wrappers), does the resulting communication mechanism still work? Does a wrapper introduce new deadlocks, failure modes or race conditions?
- Transparency: Does a wrapper change the interface of the communicating parties? Since the goal of wrappers is to avoid directly modifying the components in a system, transparency is an important feature to verify.
- Compositionality: What are the compositional and algebraic properties of a set of wrappers? This includes issues such as commutativity (can the ordering of two wrappers be exchanged?), inverses (does one wrapper undo the effects of another?), idempotence (does it matter if we apply the same wrapper twice?), and other more speci£c properties of a composition of several wrappers.

a → P	Action Pre£x
a → P | b → Q	Choice
P || Q	Parallel Composition
label:P	Process Labelling
P/{new/old}	Relabelling
P\{hidden}	Hiding
when (n<T) a → P	Guarded Action
P+ {a,b,c}	Alphabet Extension
STOP, ERROR	prede£ned processes
set S = {*a*, *b*, *c*}	de£nes a set S
range R = 0..5	de£nes a range R
[v:S]	binds variable v to a value chosen from S

Table 1. FSP quick reference

3.2. Protocol Transformation

To address questions like these, our approach is to de£ne a connector wrapper formally as a *protocol transformation*. In effect, a wrapper converts the protocol de£ning one connector into a new protocol de£ning the altered connector. The basic operations that comprise a protocol transformation may include redirecting, recording and replaying, inserting, replacing, and discarding particular events.

Building on past work in this area, we adopt an approach based on process algebras [6, 12]. Process algebras provide a way to talk about patterns of events, and are supported by a number of useful analysis tools. In particular, we use FSP[2]. (Other process algebras would have worked equally well: we chose FSP because it is simple enough for non-experts to use but can still provide a useful set of analyses, such as whether a connector protocol will deadlock or whether a safety or liveness property is violated.)

To describe a connector protocol in FSP, we use an approach similar to Wright [1]. A connector is de£ned 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 $n + 1$ processes are placed in parallel with the roles relabelled. Checks are performed on the resulting composite processes using tools such as model checkers.

Formally, for a connector with roles $R_1 \dots R_n$ and glue G, the semantics of the connector is given by Expression 1.

$$R_1 \| \dots \| R_n \| G \qquad (1)$$

[2]A quick reference for some FSP operators is given in Table 1; for further information see [11]. Processes describe actions (events) that occur in sequence, and choices between event sequences. 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 $P \| Q$, 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 are willing to engage in it.

```
Caller = (call → return → Caller).
De£ner = (call → return → De£ner).
Glue = (caller.call → de£ner.call → Glue
      | de£ner.return → caller.return → Glue).
||ProcCall = (caller:Caller || de£ner:De£ner || Glue).
```

Figure 1. Simple procedure call

To illustrate, Figure 1 shows a simple procedure-call connector. The connector has two roles, Caller and De£ner. Each engages in a call event followed by a return event. In the parallel process ProcCall, the role processes execute concurrently with the Glue process, synchronizing on shared events; here the event labels in each role process have been pre£xed with a label unique to that role (caller or de£ner), so each role shares events only with the glue, 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 De£ner role, and a return event at the De£ner role is followed by a return at the Caller role. ProcCall can then be checked for deadlock, without needing any speci£cation of the components whose communication it describes. (Later, the protocol of the component interfaces should be checked for conformance to the role speci£cations, which are effectively standing in for these future components.)

Our protocol transformation takes a connector of the form given in Equation 1 and, by adding and modifying processes in that connector, produces a new connector with the same general form. The chosen approach (outlined in the next section) makes it easy to compose, or chain together, several such transformations, to achieve a complex result from several simpler modular wrappers.

4. Connector Wrapping

The core of this wrapper-description technique is the interposition of a new process between a role process R and a glue process G. In an implementation this would correspond informally to the interposition of a new piece of code, the wrapper.

First, we decouple R and G from one another; formally this is done by renaming. Next, we add a new process W that synchronizes with both R and G. This wrapper process W re-links the two decoupled processes by intermediating between the original and the renamed events. W has the opportunity to redirect, record/replay, insert, replace, or discard the events communicated between R and G. W may be parameterized to facilitate reuse in a broader range of contexts.

Let $A_{RG} = \alpha(R) \cap \alpha(G)$ be the set of events shared by R and G. The £rst step is performed by renaming these events in *one* of the two processes, ensuring that the two processes no longer synchronize directly. For the second step, a new process W is placed in parallel with the role processes and glue process. W translates between the events in A_{RG}, in the alphabet of R, and their counterparts in the renamed alphabet of G. W in parallel with G can be thought of as a new composite glue process, G_W.

We can derive the semantics of the newly wrapped connector, given in Equation 2, by taking the original equation 1, adding W (a wrapper for one of the roles $R_1 \ldots R_n$), and replacing G with $f(G)$, where $f(G)$ is the relabelled version of G.

$$R_1 \| \ \ldots \| \ R_n \| \ W \| \ f(G) \qquad (2)$$

Some classes of wrappers may also increase the number of roles; this is done by introducing additional new processes with which W synchronizes:

$$R_1 \| \ \ldots \| \ R_n \| \ W \| \ f(G) \| \ R_{n+1} \ldots \| \ R_{n+k} \qquad (3)$$

4.1. Bene£ts

This approach isolates the wrapper from the rest of the original connector; though the core concept is not dif£cult to grasp, the technique can be powerful when used consistently and yields several bene£ts including composability, traceability, and reusability.

These wrapper speci£cations are readily composable; we can treat $W \| \ f(G)$ as a new *composite* glue, and apply a new wrapper W_2 to the W-wrapped connector:

$$R_1 \| \ \ldots \| \ R_n \| \ W_2 \| \ f_2(W \| \ f(G)) \qquad (4)$$

Traceability is facilitated by the decompositional structure of the transformed speci£cation, 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 dif£cult with a monolithic speci£cation of the protocol. Furthermore, the structure in the speci£cation 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 wrappers in the implementation that would contain a corresponding bug. This correspondence between wrapper processes and wrapper implementations can be enforced by implementation generation tools.

Parameterization techniques enable the construction of *reusable* wrapper speci£cations, applicable to a range of connector speci£cations. Generalization can be taken further[3], describing patterns or templates for types of wrappers in terms of their actions (redirect, record/replay, insert,

[3]Owing to space considerations, this technique will not be covered in this paper.

```
Caller = (call → return → Caller).
De£ner = (call → return → De£ner).
Glue = (caller.call → de£ner.call→ Glue
        | de£ner.return → caller.return → Glue).
Wrap =  (caller.call → wrap.caller.call → Wrap
         | wrap.caller.return → caller.return → Wrap).
||WrapPC =  (caller:Caller || de£ner:De£ner
              || Glue/{wrap.caller/caller} || Wrap).
```

Figure 2. "No Effect" wrapper

replace, or discard) on events; this enables automatic generation of instances of some kinds of wrapper speci£cations given a connector speci£cation and a set of inputs including the wrapper template and the affected elements of the connector.

We will illustrate the basic structure of wrapper application with three brief examples. These examples will then be used to show composition of wrappers (section 6). Then we will demonstrate parameterization of a wrapper and apply it to a different connector protocol (section 7).

5. Examples

We begin with a trivial wrapper to illustrate the basic structure of its application to a connector speci£cation. Then we introduce a connector with two classes of faults and describe two fault-tolerance wrappers.

5.1. First Wrapper: "No Effect"

Figure 2 shows the representation of a wrapper that does nothing. Such wrappers can be generated automatically from the FSP representation of a connector. Though trivial, this example provides a starting point for the construction of more complex wrappers.

The intermediary process, Wrap, simply relays events from the Caller role to the Glue role and vice versa. The new Wrap is added to the composite process WrapPC, and all events in the glue that begin with the label caller are relabelled to begin with wrap.caller so that the glue and the caller no longer synchronize directly.

If the wrap events in WrapPC are hidden from observers by means of the hiding operator, and the labelled transition system drawn by LTSA (Labelled Transition System Analyzer, a tool provided with FSP) is compared to the depiction of the original connector, it becomes apparent by inspection that the two are equivalent. Equivalence can be checked more formally by treating one version as a safety property of the other version.

```
Caller = (call → TryCall),
        TryCall = (return → Caller | err → Caller).
De£ner = (call → return → De£ner | crash → END) \{crash}.
Glue  =  (caller.call → TryCall
          | de£ner.return → caller.return → Glue),
       TryCall =  (de£ner.call → Glue | caller.err → Glue).
||FaultyRPC =  (caller:Caller || de£ner:De£ner || Glue).
```

Figure 3. Procedure call with timeouts

5.2. Another Connector

The remaining examples of wrappers will embody common dependability enhancements. In order to illustrate their application, we introduce a connector in which the caller occasionally receives errors (Figure 3).

The FaultyRPC connector is subject to timeout errors that can occur whenever the caller is attempting to contact the de£ner.[4] 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 de£ner component.[5]

The task of a dependability-enhancing wrapper for this connector is to hide the errors from the caller: it must be possible for a err-unaware caller not merely to avoid deadlock but also to make progress.

We will apply two wrappers to this connector: one that hides transient faults, and one that hides the failure of the de£ner. Transient faults can be masked by re-sending the request that had timed out. (This technique is in common practice; examples, such as retransmission in TCP [16], abound.) To mask the component failure, one possible technique is to provide a more reliable[6] backup, 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 [10].)

5.3. Second Wrapper: "Retry"

We begin with a wrapper that will retry inde£nitely, and then show how it may be revised to retry only a £nite number of times. The wrapper intercepts any timeout error sent

[4]Timeouts when the de£ner is replying to the caller can also be modelled by adding another choice branch to the glue.

[5]The possibility of subsequent recovery of the failed component can be modelled, but is not included in this example for simplicity. Also, although a non-transient failure could also be due to a failure in the communications channel (such as being severed by a backhoe), for this example we will model only the failure of the component.

[6]It may be lacking in other key qualities such as performance, for which the primary may be generally preferred, as in [17].

to the client, and send out a new call event to the glue (to be relayed to the de£ner) instead: see Figure 4.

The structure of the composite RetryRPC is essentially identical to that of the WrapPC already seen (Figure 2). The RetryAll process is very similar to the previous no-effect Wrap process, with the addition of a new choice branch triggered by the retry.caller.err sent by the glue.[7]

This wrapper illustrates interception and replacement of events without change to the interfaces of the connector[8]. Wrappers can also enable the addition of new participants (roles) to the communication, as shown in a subsequent example.

5.3.1 Checking Results

Having applied this wrapper to our faulty connector, we would like to know several things: Is the result sound, or will it deadlock? Is the wrapper transparent to the caller role, or has it changed the interface? Have errors been hidden from the caller role? Does the caller role always make progress, or (as we expect) can the system become wedged: unable to make progress? We can use the LTSA model checker to con£rm that RetryRPC is deadlock-free as well as to answer the remaining questions.

FSP allows us to de£ne "safety properties" that constrain event ordering of legal events and prohibit illegal events. The £rst part of a safety property is a *process* de£ning the legal event ordering; the second part of a safety property is a *set* of any events that should be considered illegal. FSP also allows us to de£ne "progress properties"; a progress property speci£es a *set* of desirable events, and the system is considered to be making progress when at least one of these events occurs in£nitely often. Safety and progress properties can be checked with LTSA.

To con£rm that the caller's interface need not change, we restate the original caller role as an FSP safety property. The "process" part of this safety property would be the caller role; we'll leave empty the "set" part of this safety property. LTSA will show a safety violation and event trace, if the legal event ordering is not followed.

To con£rm that errors will not reach the caller role, we write another safety property: the NoErrors property shown in Figure 5. Here, event ordering is unconstrained (the "process" part of the property is simply STOP), and there is one

[7]Wrapper processes such as this one may also be generated automatically using tools that we are developing; £rst by generating the no-effect wrapper particular to the desired connector and then by modifying the no-effect wrapper according to a connector-independent template, e.g. adding a choice branch as seen here.

[8]In the case of in£nite Retry, one might *choose* to replace the original role R_0 with a more speci£c interface R_1 that does not engage in error events; in£nite Retry is nevertheless a transparent addition for our purposes, because the event traces accepted by R_1 are a *subset* of those that R_0 accepts, so a component that was compatible with R_0 will still be compatible with R_1.

```
Caller  = (call → return → Caller).
De£ner =  (call → return → De£ner
              | crash → END) \{crash}.
Glue =  (caller.call → TryCall
            | de£ner.return → caller.return → Glue),
        TryCall =  (de£ner.call → Glue | caller.err → Glue).
RetryAll =  (caller.call → retry.caller.call → RetryAll
               | retry.caller.return → caller.return → RetryAll
               | retry.caller.err → retry.caller.call → RetryAll).
||RetryRPC =  (caller:Caller || de£ner:De£ner
                 || Glue/{retry.caller/caller} || RetryAll).
```

Figure 4. Applying RetryAll

```
property NoErrors = STOP + {caller.err}.
progress CallerOk = {caller.return}
||RetryRPC =  (caller:Caller || de£ner:De£ner
                 || Glue/{retry.caller/caller} || Retry
                 || NoErrors ).
```

Figure 5. Safety and progress

```
Caller = (call → return → Caller).
De£ner =  (call → return → De£ner | crash → END) \{crash}.
Glue =  (caller.call → TryCall
            | de£ner.return → caller.return → Glue),
        TryCall =  (de£ner.call → Glue | caller.err → Glue).
const T = 3
Retry = Retry[0],
  Retry[n:0..T] = (caller.call → retry.caller.call → Retry[0]
  | retry.caller.return → caller.return → Retry[0]
  | when (n<T) retry.caller.err → retry.caller.call → Retry[n+1]
  | when (n==T) retry.caller.err → caller.err → Retry[0]).
||RetryRPC = (caller:Caller || de£ner:De£ner
                 || Glue/{retry.caller/caller} || Retry).
```

Figure 6. Applying Retry

illegal event, caller.err. LTSA will show a safety violation and event trace, if an illegal event can occur.

To con£rm that the caller will *make progress* in the case of a failed de£ner, we de£ne the progress property CallerOk. Its set of desirable events contains simply caller.return. In this example, just as we expect, LTSA reports a progress violation and the trace shows the failure of the de£ner.

5.3.2 Revising the Wrapper

Figure 6 shows a revised wrapper that re-sends at most 3 times in a row, using a parameterized local process that is incremented on each retry (and is reset on each non-error branch). It can be substituted for the RetryAll wrapper previously shown.

```
Caller = (call → return → Caller).
De£ner = (call → return → De£ner | crash → END) \{crash}.
Glue = (caller.call → TryCall
        | de£ner.return → caller.return → Glue),
    TryCall = (de£ner.call → Glue | caller.err → Glue).
Backup = (call → return → Backup).
BGlue = caller.call → de£ner.call → BGlue
        | de£ner.return → caller.return → BGlue).
Failover = (caller.call → pri.caller.call → Failover
            | pri.caller.return → caller.return → Failover
            | pri.caller.err → bk.caller.call → ToBk),
  ToBk = (caller.call → bk.caller.call → ToBk
            | bk.caller.return → caller.return → ToBk).
||FailoverRPC = (caller:Caller || pri.de£ner:De£ner || pri:Glue
    || bk.de£ner:Backup || bk:BGlue || Failover).
```

Figure 7. Applying Failover

Retransmission masks one of the two kinds of errors to which the FaultyRPC connector is subject. The £nite-retry wrapper produces a wrapped connector that is subject only to errors that are (probably) symptoms of a failed de£ner. We show later how this connector can be wrapped in turn to mask de£ner failure. Such chaining is common practice in fault tolerance, to achieve a greater coverage of possible failure situations by using a combination of several techniques drawn from broad classes (such as detection, diagnosis, containment, masking, compensation, and repair [5]; classi£cations vary but in general are ordered from the less drastic, lighter weight to the last-ditch, heavier weight techniques and are used in that order). This practice further motivates our compositional approach.

5.4. Third Wrapper: "Failover"

In order to mask the potential failure of the de£ner, we introduce component redundancy, which requires adding a new participant in the communication. The wrapper in this example (Figure 7) triggers the switch from primary to backup and performs the subsequent redirection of calls from the caller component to the backup. The local process ToBk, de£ned within the Failover process, redirects calls to the backup.[9]

[9]Note that we have modeled the simple case in which the primary never recovers. Slight modi£cation of Failover would be required to handle the case in which the primary may eventually recover: just as the Failover process has a trigger to switch to ToBk, ToBk could be given a trigger to switch back to the primary, or if the primary's recovery is also silent, ToBk could employ a counter (similar to the Retry process) and periodically attempt use of the primary.

```
||ReOver = (caller:Caller || pri.de£ner:De£ner
    || pri:( Glue/{retry.caller/caller} || Retry )
    || bk.de£ner:Backup || bk:BGlue || Failover).
```

Figure 8. Composing Retry and Failover

```
property NoErrors = STOP + {caller.err}.
progress CallerOk = {caller.return}
||ReOver = (caller:Caller || pri.de£ner:De£ner
    || pri:( Glue/{retry.caller/caller} || Retry )
    || bk.de£ner:Backup || bk:BGlue
    || Failover || NoErrors ).
```

Figure 9. Safety and progress, revisited

```
||ReReOver = (caller:Caller || pri.de£ner:De£ner
    || pri:( Glue/{retry.caller/caller} || Retry )
    || bk.de£ner:Backup
    || bk:( Glue/{retry.caller/caller} || RetryAll )
    || Failover).
```

Figure 10. Retry and RetryAll and Failover

6. Composition

Now we have wrappers that embody two reliability-enhancing techniques, each suited to one of the failure modes of the original faulty connector. Naturally we wish to compose the two wrappers. Our incremental approach makes this straightforward. Had we taken another approach to specifying wrapped connectors, such as simply modifying the existing glue processes rather than placing a new intermediary process beside them, the results of applying a single wrapper might have appeared simpler, but composition of wrappers would be quite dif£cult.

To compose these two wrappers, the reference to the Glue process in FailoverRPC is replaced with the enhanced "glue" of RetryRPC, as in Figure 8 (the elements that came from RetryRPC's "glue" are shown in bold font).

These two wrappers are *not commutative* (readily evident as the replacement of the Glue in RetryRPC with the enhanced "glue" of FailoverRPC yields a non-equivalent state machine in LTSA), thus it is important to get the order of application right; fortunately it is fairly intuitive to translate the desired effect "try x; if that's not enough, try y" into the right order "wrap with x, then wrap the composite with y".

Consider again the questions posed in section 5.3.1. By reusing the NoErrors and CallerOk safety and progress properties, as shown in Figure 9, we con£rm that errors are still hidden from the caller role, and furthermore that the caller *will* now make progress.

Finally, this example assumes no transmission dropouts between the caller and the backup de£ner (BGlue will not

```
set C = {caller}
set D = {de£ner}
set COut = {call}
set CIn = {return}
set NewLabel = {wrap}
Wrap = ([r:C].[e:COut] → [NewLabel].[r].[e] → Wrap
        | [NewLabel].[r:C].[e:CIn] → [r].[e] → Wrap).
||WrapPC = (caller:Caller || de£ner:De£ner
           || Glue/{[NewLabel].[r:C]/[r]} || Wrap).
```

Figure 11. Parameterized "No Effect"

generate errors on its own). To eliminate the unrealistic BGlue, we can replace it with an unreliable glue that has been wrapped with the RetryAll wrapper (Figure 10; the elements that came from RetryAll's "glue" are shown in bold font).

7. Parameterizing for Reuse

The wrapper processes in the preceding examples are hardcoded for the speci£c connector that they wrap. It is preferable to write more generalized and reusable wrapper processes via parameterization, so that they can be applied in a different context to a different connector type. We now see how to generalize the earlier examples to accomplish this in a straightforward way, and apply the result to a different connector.

7.1. Revisiting "No Effect"

Figure 11 shows how generalization can be performed for the simple "no effect" wrapper. To apply the wrapper to a connector, we must £ll in the italicized regions, de£ning 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 Wrap process, values of variables are drawn from these sets and are bound within a sequence.

For example in the £rst line of Wrap, for the £rst event, any value of r drawn from C is acceptable, and any value of e drawn from COut. In this application, each set has only one element, so the only corresponding event is caller.call. The next event in the sequence will pre£x the [r].[e] event (bound to caller.call) with a label drawn from the one-element set NewLabel. The result is wrap.caller.call.

Similarly, in the composite process WrapPC, the relabelling of the Glue can be parameterized to ensure it matches the labels used in the Wrap process; the label drawn from NewLabel will be added to the beginning of each event pre£xed with any label in C. The £nal lines of the £gure

```
set C = {caller}
set COut = {call}
set CIn = {return}
set CErr = {err}
set L = {retry}
set CInit = {call}
const T = 3
Retry = 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 || de£ner:De£ner
             || Glue/{[L].[r:C]/[r]} || Retry).
```

Figure 12. Parameterized Retry)

show the pattern for the composite process of a wrapped connector that uses this wrapper; the italicized roles should be £lled in with the names of the actual roles.

7.2. Revisiting "Retry"

Parameterization of the Retry wrapper, shown in Figure 12, is similar. This wrapper must remember which event the caller attempted to transmit, so that that speci£c event can be repeated. Note the set CInit, used to initialize the cached event. Due to limitations of FSP syntax, a hidden event (hide) 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).

7.3. Applying to a New Connector

Figure 13 shows a new example connector. Here operation must begin with an open event, and end with a close event. During normal operation the client makes requests, and the server responds with a range of numeric values. Only the server can choose the returned value, and only the client can choose when to close. Timeouts (shown in bold font) may occur when the client is attempting to send requests or to close the connection. This connector *will deadlock*, since the client role is not expecting an error. The deadlock is eliminated when we apply the parameterized retry wrapper of Figure 12.

```
range M = 0..5
Client = (open → Run),
  Run = (request → result[v:M] → Run | hide → Close),
  Close = (close → END) \{hide}.
Server = (open → Run),
   Run = (request → hide[e:M] → result[e] → Run
          | close → END) \{hide}.
Glue = (client.open → server.open → Glue
        | client.request → server.request → Glue
        | server.result[v:M] → client.result[v] → Glue
        | client.close → server.close → END).
        | client.request → client.err → Glue
        | client.close → client.err → Glue
||Conn = (client:Client || server:Server || Glue).
```

Figure 13. Another connector

```
set C = {client}
set COut = {open, request, close}
set CIn = {result[v:M]}
set CErr = {err}
set L = {retry}
set CInit = {open}
||RetryRPC = (client:Client || server:Server
              || Glue/{[L].[r:C]/[r]} || Retry).
```

Figure 14. Applying parameterized Retry

Figure 14 shows the elements that must be £lled in to apply the Figure 12 wrapper to the Figure 13 connector.

The Retry pattern can also be applied to connectors with more than two roles, such as a client-server connector with multiple client roles. However, if the glue process constrains the behavior of the role processes (perhaps by allowing or disallowing nesting of calls), the wrapper process must cooperate in enforcing this constraint. In the approach we have been using this is achieved by patterning the wrapper on the results of exposing only a subset of events (those engaged in by the roles that will be adjacent to the wrapper) in the glue process, similar to the "no effect" wrapper.

7.4. Detecting Errors

Mistakes in the construction and application of parameterized wrappers can still be caught by the safety and progress analyses described earlier. This becomes increasingly useful as wrappers are composed and the speci£cation size increases.

For example, Figure 15 shows a plausible mistake in the application of a parameterized Failover. The event err (shown in bold font) is listed in the set of events that are acceptable as input to the caller; this would allow some error events to be relayed to the caller, at the whim of the Failover process. This mistake is caught by the NoErrors property (used to check that the wrapper does its intended job, as seen earlier in Figure 5), no matter whether the caller process accepts err, as in Figure 3, or ignores err, as in Figure 7. Once diagnosed, the problem can be resolved by removing err from CIn.

```
set C = {caller}
set COut = {call}
set CIn = {return, err}
set CErr = {err}
Caller = ...
De£ner = (call → return → De£ner | crash → END) \{crash}.
Glue = (caller.call → TryCall
        | de£ner.return → caller.return → Glue),
     TryCall = (de£ner.call → Glue | caller.err → Glue).
Backup = (call → return → Backup).
BGlue = (caller.call → de£ner.call → BGlue
         | de£ner.return → caller.return → BGlue).
Failover = ([r:C].[e:COut] → pri.[r].[e] → Failover
            | pri.[r:C].[e:CIn] → [r].[e] → Failover
            | pri.[r:C].[CErr] → bk.[r].call → ToBk),
  ToBk = ([r:C].[e:COut] → bk.[r].[e] → ToBk
            | bk.[r:C].[e:CIn] → [r].[e] → ToBk).
||FailoverRPC = (caller:Caller || pri.de£ner:De£ner || pri:Glue
      || bk.de£ner:Backup || bk:BGlue || Failover).
```

Figure 15. Parameterized Failover

7.5. In Review

The initial examples of wrappers were hard-wired and could only be used for one particular connector speci£cation. In implementation, this would correspond to particular pieces of code that could be applied in systems where a particular connector implementation is used (such as a Java RMI connector, an example of an RPC-style connector), but could not be applied to some other arbitrary connector implementation (such as a Unix pipe, or even a different implementation of RPC).

Here in section 7 we have shown how to write parameterized wrappers that are applicable to, and reusable across, *multiple* connector types. This generalization is straightforward to accomplish. The parameterized wrappers are not dif£cult to apply to connector speci£cations. Furthermore, even if a mistake is made, it can be readily detected (as shown in 7.4).

In implementation, this generalization into parameterized wrapper patterns, which are then applied to particular connector types, could correspond to the use of a tool or "wizard" that is prompted with a wrapper pattern plus a small amount of information about a connector, This tool would then generate an instance of a wrapper implementation suitable for that particular connector implementation (which may be a CORBA connector, a Java Message Service connector, etc.). We use this approach in [20].

8. Discussion

An important consideration when producing a formal speci£cation, or a family of formal speci£cations (as we have proposed), is the resulting engineering properties of the formal artifact. Our approach has two important properties: *compositionality* and *traceability*. Here compositionality refers to the ability to combine wrappers to create a more complex composite wrapper, as in Figure 8. Traceability refers to the ability to determine where something "came from", so that if a problem is discovered, its source can be located in a particular section of the model, and the corresponding affected section of the implementation can be determined, and vice versa.

As we have demonstrated in the example of section 6, this wrapper speci£cation technique exhibits ease of composition. The effects of event redirection, insertion, replacement, deletion, etc., are achieved by interposing a new process, rather than by actually editing an existing process. As a result composition of the wrappers is straightforward. To apply an existing wrapper it is only necessary to classify events into a small number of sets and to perform a renaming on the glue. (It is naturally also easy to remove a wrapper.) If layers of enhancement were added instead by, for example, performing directives that state how to mutate the glue process into a monolithic enhanced glue process (such as, after each event e of type T, add a new event $f(e)$), either automatically or by hand, the result would become increasingly dif£cult to modify further, and removal of an arbitrary enhancement would not be straightforward.

Traceability between the protocol speci£cation and the implementation is promoted by the essential similarity of their respective structures. When wrappers are actually implemented, it is generally as a layer of enhancements interposed between the component interface and the communication infrastructure; this interposition may be supported by interceptors or system-level trickery, but in any case leaves the component and the infrastructure essentially undigested and unchanged. By using a similar wrapping technique in the protocol speci£cation, its structure remains similar to the structure of the implementation (implementation wrappers correspond to wrapper processes in the speci£cation), enabling the tracing of attributes from a substructure of one to the corresponding substructure of the other.

It is worth noting that, while there are many contexts in which wrappers must ideally achieve a transparent effect, in other cases it is desirable for information available to one wrapper to be exposed to a subsequent wrapper (when ordinarily this information would deliberately be concealed). Often this can be achieved, without adversely impacting any intervening wrappers, by using parameterization that is already in place. For example[10], consider a system with three high-performance De£ner components and one slow but reliable Backup component. We apply the parameterized Retry and Failover wrappers to each of three FaultyRPC connectors. Now we wish to apply a new LoadBalancer wrapper which, given a caller.call event, redirects it to *one* of the three wrapped connectors and thus, ultimately, to either the corresponding De£ner or the single Backup component. Clearly it would be wise to expose the failure of a component to LoadBalancer so that it has the opportunity to refrain from using it. This can be done by adding a tag to [r].[e] in the Failover process, and extending the set CIn of any wrappers between Failover and LoadBalancer to include these tagged events.

Finally, one important question is, can this formalism correspond to implementation as hinted at in section 7.5? To what extent can it be related back to the "real world"? To answer this question is beyond the scope of this paper, but in other work [20], transformation patterns are encoded as operations on stub generation tools. For example we have shown how to create and apply transformation patterns that are very similar to the kinds of wrappers described here. Using the stub generation tools, implementations of these transformations have been generated for Java RMI, a commonly used RPC-style connector implementation, and this work has been extended to Java Message Service, an event-style connector.

9. Conclusion

This work provides a formal framework for reasoning about wrappers and their effect on interaction mechanisms via protocol enhancement. We have illustrated the use of this technique for dependability-oriented wrappers and shown how compositionality is achieved, and how analyses may be used to con£rm desired attributes, such as whether a wrapper preserves existing interfaces while also masking communication errors. This approach allows practitioners to break a complex modi£cation into incremental, more easily understood parts, and reason about wrappers' effects in advance of their implementation.

More generally, this work also provides an example of a formal abstraction or model that has a good engineering basis, providing not just a means of principled reasoning, but one that also has an increased degree of compositionality, checkability, traceability, and maintainability.

An additional contribution of this work is to demonstrate usefulness of the Wright approach, which separates a connector into interfaces and their interactions.

[10]Thanks are due to an anonymous reader of an earlier draft who suggested this scenario

Several questions were posed in section 3. We have addressed a number of them, showing how to answer them for at least the speci£c examples shown.

- Soundness: Given a connector and a wrapper, we have shown how to construct a wrapped connector, on which a model checker can be used to determine whether the wrapper introduces new deadlocks.
- Transparency: The interfaces of a wrapped connector can be checked to ensure conformance.
- Compositionality: We show how to compose several types of wrappers and reason about that composition, including determination of noncommutativity. Checking other algebraic properties, such as idempotence and inverses, can be addressed in a manner similar to transparency conformance checks.

Work is ongoing to complete a set of genericized protocol transformation patterns comparable, and complementary, to the common patterns of connector implementation enhancement described in other work [20]. The examples given here illustrate the rudiments of the approach, giving semantics for a subset of the enhancement patterns, and thus supporting increased understanding of the results of their application and composition. Our other wrapper speci£cations (not included here due to space) include compression, encryption, logging, authorization, and voting wrappers, dynamic protocol selection and a limited set of protocol mismatch-repair wrappers.

10. Acknowledgments

This research has been supported by NSF under grant CCR-0113810, by DARPA under contract F30602-00-2-0616, and by NASA under the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the of£cial policies, either expressed or implied, of NSF, DARPA, NASA, or the U.S. government. We would like to thank Philip Koopman, Mary Shaw, Jeannette Wing, and the members of the ABLE group.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] R. DeLine. *Resolving Packaging Mismatch*. PhD thesis, Carnegie Mellon, School of Computer Science, 1999. Issued as CMU Technical Report CMU-CS-99-141.

[3] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[4] D. Garlan, R. Allen, and J. 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.

[5] W. L. Heimerdinger and C. B. Weinstock. A conceptual framework for system fault tolerance. Technical Report CMU/SEI-92-TR-33, Carnegie Mellon University, 1992.

[6] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[7] H. Hueni, R. E. Johnson, and R. Engel. A framework for network protocol software. *Proceedings of OOPSLA'95*, pages 358–369, 1995.

[8] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, WA, July 1999.

[9] A. Lopes, M. Wermelinger, and J. L. Fiadeiro. A compositional approach to connector construction. In *Recent Trends in Algebraic Development Techniques*, volume LNCS 2267, pages 201–220. Springer-Verlag, 2001.

[10] M. R. Lyu. *Software Fault Tolerance*. John Wiley and Sons, 1995.

[11] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 1999.

[12] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[13] P. Narasimhan, L. E. Moser, and P. M. 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.

[14] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[15] J. Pan, P. Koopman, D. Siewiorek, Y. Huang, R. Gruber, and M. L. Jiang. Robustness testing and hardening of CORBA ORB implementations. In *Proceedings of the International Conference on Dependable Systems and Networks (ICDSN/FTCS)*, pages 141–150, July 2001.

[16] J. Postel. Transmission control protocol. Technical report, RFC-793, 1981.

[17] L. Sha, J. Goodenough, and B. Pollack. Simplex architecture: Meeting the challenges of using COTS in high-reliability systems. *Crosstalk*, April 1998.

[18] M. Shaw. Procedure calls are the assembly language of system interconnection: Connectors deserve £rst-class status. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[19] M. 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.

[20] B. Spitznagel and D. 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.

[21] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. Technical report, Cornell/TR97-1638, 1997.