Submitted for publication.

Towards a Formalization of Connector Wrapping

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 (or 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 certain extra-functional deficiencies.

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 difficult 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 be not feasible, since components are often built by third parties, or are sufficiently 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 using 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 detecting and rejecting illegal input, allowing only well-tolerated inputs to reach the component, and thereby increasing the reliability of the system as a whole [14].

Unfortunately the current practice of wrapper creation and use is ad hoc and something of a black art. To overcome a (perhaps unforeseen) difficulty quickly, a one-off wrapper is written specifically 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 specific wrapper, interposed between a component and a connector, affect the protocol of that connector? Are existing properties of the protocol maintained, or gone; 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 modifications 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 specification 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 specification to an *implementation* using tools and techniques that help enforce that the implementation corresponds to the specification.

In this paper we address the first two of these issues for an important subclass of wrappers, namely *connector wrappers*. As we detail later, these wrappers are used to repair or augment communication-related properties of a system, such as the two examples mentioned above.[1] 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 specifications, demonstrate the natural compositionality of this kind of wrapper specification, and illustrate how analyses can be used to confirm whether the wrappers actually achieve their intended purpose.

2. RELATED WORK

The work is motivated in part by the observation that wrapper use is widespread: sufficiently widespread as to have given rise to efforts directed at standardized support for wrapper introduction. System-level support mechanisms (usually called *interceptors*) have become increasingly available for implementations of some commonly used connector types. Interceptors facilitate the insertion of arbitrary application-level wrapper code. Wrappers introduced via interceptors may be used to enhance fault tolerance [12] and security [3] of COTS components, or to add instrumentation [8]. However, by their nature these efforts are specific 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, 11]. In particular, it applies to FSP [10] 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. We also make use of the idea, drawn from software architecture, of treating a software connector as a separate first-class entity [17], on a par with software components.

Some work in software architecture strives to mitigate, in a different way, the kinds of problems that give rise to wrappers. 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 [18]. 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 flexible packaging approach is elegant but has not been widely adopted by component providers, thus is unlikely to replace practitioners' use of wrappers in the immediate future.

The area of protocol synthesis is also related to the work described in this paper, in that it deals with protocols' composition from (or decomposition into) simpler protocols. Ensemble [20], enables the construction of an adaptive protocol composed of stacked micro-protocol modules. The x-Kernel [13] 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+ [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 flow 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.

3. OVERVIEW OF APPROACH

As discussed earlier, informally a *wrapper* is new code interposed between component interfaces and infrastructure support. 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 specific 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.

Our goal is to provide a more formal, disciplined approach to

[1] We address the third issue in [19].

connector wrapper design (and indirectly implementation), so that we can understand their behavior and other properties. Specifically, 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 specific properties of a composition of several wrappers.

To address questions like these, our approach is to define a connector wrapper formally as a *protocol transformation*. That is to say, the effect of a wrapper is to convert the protocol defining one connector into a new protocol defining the altered connector. The basic operations that may be used in such a protocol transformation include redirecting, recording and replaying, inserting, replacing, and discarding particular events.

More specifically, building on past work in this area, we adopt an approach based on process algebras [6, 11]. Process algebras provide a way to talk about patterns of events, and are supported by a number of useful analysis tools. In particular, we will 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 hand-written safety or liveness property is violated.)

When describing a connector protocol in FSP we use an approach similar to Wright [1]. 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 $n + 1$ processes are placed in parallel with the roles relabelled. Checks (e.g., for deadlock) can be performed on the resulting composite processes using tools such as model checkers.

[2] A quick reference for some FSP operators is given in Table 1; for further information see [10]. 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.

a → P	Action Prefix
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

Table 1: FSP Quick Reference

```
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 1: Simple Procedure Call

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

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

Figure 1 illustrates the idea with a simple procedure-call connector. The connector has two roles, Caller and Definer. 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 prefixed with a label unique to that role (caller or definer), 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 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 specification of the components whose communication it describes. (Later, the protocol of the component interfaces should be checked for conformance to the role specifications, which are effectively standing in for these future components.)

Our protocol transformation will take a connector of the form given in Equation 1 and, by adding and modifying processes in that connector, produce a new connector with the same general form. One of the more interesting aspects of the particular approach chosen (which will be outlined in the next section) is that it is very 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 the implementation this corresponds informally to the interposition of a new piece of code, the wrapper implementation.) This technique has two steps. First, decouple R and G from one another; formally this is done by renaming. Second, add a new process W that will synchronize with both R and G. This wrapper process W re-links

```
Caller = (call → return → Caller).
Definer = (call → return → Definer).
Glue = (caller.call → definer.call→ Glue
        | definer.return → caller.return → Glue).
Wrap = (caller.call → wrap.caller.call → Wrap
        | wrap.caller.return → caller.return → Wrap).
||WrapPC = (caller:Caller || definer:Definer
           || Glue/{wrap.caller/caller} || Wrap).
```

Figure 2: Wrap with No Effect

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.

Let $A_{RG} = \alpha(R) \cap \alpha(G)$ be the set of events shared by R and G. The first 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 the renamed versions of these events, in the alphabet of G. For the purpose of composition, W in parallel with G can be thought of as a new composite glue process, G_W, to which other wrappers might be applied in future.

We can derive the semantics of the newly wrapped connector, given in Equation 2, by taking the original equation 1, adding W, and replacing G with G', where G' is the relabelled version of G.

$$R_1 || \ldots || R_n || W || G' \tag{2}$$

We will illustrate this idea with several examples: an "identity" wrapper that has no effect; a "retry" wrapper that retransmits events when an error is detected; and a "fail-over" wrapper that redirects communication to a backup role upon detecting failure of the original role.

4.1 Wrap with No Effect

The base case for our approach is to show how to represent a wrapper that does nothing. Although not of practical value it shows in the simplest form how wrappers are used, and provides the starting point for introducing more complex transformations.

Figure 2 shows how this wrapper is specified.[3] 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 (now called 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.

[3]Unchanged parts of the original connector will be shown in smaller text.

```
Caller = (call → TryCall),
      TryCall = (return → Caller | error → Caller).
Definer = (call → return → Definer
          | crash → END) \{crash}.
Glue = (caller.call → TryCall
       | definer.return → caller.return → Glue),
     TryCall = (definer.call → Glue
               | caller.err → Glue).
||FaultyRPC = (caller:Caller || definer:Definer || Glue).
```

Figure 3: Procedure Call with Timeouts

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, as will be discussed later.)

4.2 A Protocol That Includes Faults

The simple example of Figure 1 is a good start, but in order to illustrate a means of increasing the dependability of a connector, we will need an example that is not completely dependable! In this extended example we introduce a connector in which the caller occasionally receives errors. This connector can be wrapped so that the caller does not see errors (i.e. has the same role interface as the original example of Figure 1) and is able to make progress (i.e. receives a return event infinitely often); this example will show what such wrappers would look like using this technique and how to check that the purpose of the wrappers is achieved.

The connector depicted in Figure 3 is similar to the original example (Figure 1), but is subject to timeout errors that can occur whenever the caller is attempting to contact the definer.[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 definer component.[5]

4.3 Retransmission

Transient faults can be masked by re-sending the request that had timed out. (This is a standard technique, in common practice; examples, such as retransmission in TCP [15], abound.) A naive approach will be illustrated first: whenever a timeout error would be passed to the client, intercept it and send out a new call event to the glue (to be relayed to the definer) instead.

[4]Timeouts when the definer 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.

```
Caller = (call → return → Caller).
Definer =  (call → return → Definer
             | crash → END) \{crash}.
Glue =  (caller.call → TryCall
          | definer.return → caller.return → Glue),
       TryCall =  (definer.call → Glue
                    | caller.err → Glue).
Retry = (caller.call → retry.caller.call → Retry
          | retry.caller.return → caller.return → Retry
          | retry.caller.err → retry.caller.call → Retry).
||RetryRPC = (caller:Caller || definer:Definer
                || Glue/{retry.caller/caller} || Retry).
```

Figure 4: Procedure Call with Retry (Naive)

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

Figure 5: Safety and Progress

Figure 4 illustrates the naive "perpetual retry." The structure of the composite RetryRPC is essentially identical to that of the WrapPC already seen (Figure 2). The Retry process is identical 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.

We can formally confirm that errors will now never reach the caller role. To do this we replace the error-aware Caller process (seen in the FaultyRPC example) with the original caller role, and then use LTSA to show that the connector will not deadlock. More generally, the assertion that a caller role will see no errors can be checked by evaluating a "safety property". The NoErrors safety property shown in Figure 5 has two parts: first, a process that constrains event ordering of legal events (in this example, they are unconstrained[6]); second, a set of illegal events (in this case, only the event caller.err). If an illegal event occurs, this property will force a transition to the ERROR state and the violation will be detected. This check is easily automated using the LTSA model checker.

However, one might also like to check that the caller will *make progress* in the case of a failed definer. We can define a progress property as a set of events such as CallerOk; the system is making progress when at least one of these events is occurring infinitely often. As one might expect for this example, when we check this progress property using LTSA, we find that it is violated. The system can reach a state in which the caller does not make progress.

[6] Rather than leaving it unconstrained, we might instead have copied the process that represented the original connector's caller role, to confirm that the wrapped connector's caller interface remains unchanged from the calling component's perspective.

```
Caller = (call → return → Caller).
Definer =  (call → return → Definer
             | crash → END) \{crash}.
Glue =  (caller.call → TryCall
          | definer.return → caller.return → Glue),
       TryCall =  (definer.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 || definer:Definer
                || Glue/{retry.caller/caller} || Retry).
```

Figure 6: Procedure Call with Retry

4.4 Bounded Retransmission

Perpetual retry is not a very sensible solution for this particular connector, although it is useful for illustration. Once the definer fails, the connector will be stuck in a loop in which the caller will make no progress. If, instead, this wrapper would attempt the retry only a few times in a row, and then let the error pass through (with an implicit diagnosis of definer failure), there is the opportunity for either the caller or another wrapper to take a more suitable action. Such chaining is common practice in fault tolerance, to achieve a greater coverage of possible failure situations by utilizing a combination of several techniques drawn from broad classes (such as detection, diagnosis, containment, masking, compensation, and repair [5]; classifications vary but in general are ordered from the less drastic, lighter weight to the last-ditch, heavier weight techniques and are utilized in that order).

Figure 6 shows a new wrapper that will retry up to 3 times in a row using a parameterized local process that is incremented on each retry (and is reset on each non-error branch). Note that the new version of the Retry process can be inserted in place of the original perpetual Retry without any further changes to the connector.

The Retry wrapper is an example of a wrapper in which one event is intercepted and replaced by another event without change to the interfaces (roles) of the connector. Wrappers can also enable the addition of new participants to the communication as will be apparent in the next example.

4.5 Fail-over to Backup

The naive Retry was replaced by a counting Retry in the hopes that someone downstream would be better equipped to handle the apparent failure of the definer component. One possible technique is to provide a more reliable[7] backup, which will take over for the primary definer when a failure

[7] It may be lacking in other key qualities such as performance, for which the primary may be generally preferred, as in [16].

```
Caller = (call → return → 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 = (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.definer:Definer || pri:Glue
    || bk.definer:Backup || bk:BGlue
    || Failover).
```

Figure 7: Procedure Call with Fail-over

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 [9].

The introduction of component redundancy will require adding a new participant in the communication. A wrapper will trigger the switch from primary to backup and will perform the subsequent redirection of calls from the caller component to the backup. Such a wrapped connector is shown in Figure 7. The local process ToBk, defined within the Failover process, redirects calls to the backup.[8]

4.6 Composition

Now we have wrappers that embody two reliability-enhancing heuristics, each suitable to one of the failure modes of the original faulty connector (transient problems and silent failure of definer). It would naturally be desirable to compose the two wrappers, so that first one heuristic and then the other will be employed in the case of persistent errors.

Thankfully 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 to a trivial example might have appeared simpler, but composition of wrappers would be quite difficult.)

To compose these two wrappers, the reference to the Glue process in FailoverRPC will be replaced with the enhanced "glue" of RetryRPC, as in Figure 8.

[8] Note that we have modeled the simple case in which the primary never recovers. Slight modification 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 second Retry process) and periodically attempt use of the primary.

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

Figure 8: Composing Retry and Fail-over

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

Figure 9: Safety and Progress

In FSP, *process labelling* (the operation used to add the pri prefix) is performed before *relabelling* (the operation used to change the caller events to retry.caller), thus it is necessary to also distribute the pri. prefix over the caller and retry.caller events in the relabelling operation. Aside from this slight hitch, the composition of these wrappers is entirely straightforward thanks to the process-interposition approach.

We can confirm that the caller will see no errors and that the caller *will* make progress by reusing the NoErrors and CallerOk safety and progress properties defined earlier. To check the connector wrapped with retry and fail-over, we add them to the composite process as shown in Figure 9 and run LTSA analyses.

5. GENERALIZATION

The wrapper processes in all of the preceding examples are hardcoded for the specific connector that they wrap. However it is preferable to write more generalized and reusable wrapper processes via parameterization, so that they can be applicable to more than one connector type. We now see how to generalize the earlier work to accomplish this in a straightforward way and apply the result to a different connector.

5.1 Revisiting "No Effect"

Figure 10 shows how generalization can be performed for the simple "no effect" wrapper. 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 Wrap process, values of variables are drawn from these sets and are bound within a sequence.

For example in the first line of Wrap, for the first event, any value of r drawn from C is acceptable, and any value of e

```
Wrap = (caller.call → wrap.caller.call → Wrap
          | wrap.caller.return → caller.return → Wrap).
```

becomes

```
set C = {caller}
set D = {definer}
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 || definer:Definer
                || Glue/{[NewLabel].[r:C]/[r]} || Wrap).
```

Figure 10: Parameterizing Wrap

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

Figure 11: Retry (naive parameterization)

drawn from COut. In this instance each set has only one element (which need not be the case in general) so the only corresponding event is 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 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 prefixed with any label in C. The final lines of the figure show the pattern 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.

5.2 Revisiting "Retry"

Parameterization of the Retry wrapper is similar although slightly more complex. The naive approach shown in Figure 11 will achieve the desired effect in the case of this simple connector. However, for a two-role connector in which the caller role may be initiating more than one kind of event (as will be seen in Figure 13), some modification is required to produce a more generally useful wrapper. Simply put, the

```
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 || definer:Definer
                || Glue/{[L].[r:C]/[r]} || Retry).
```

Figure 12: Retry (improved parameterization)

wrapper must remember which event the caller attempted to transmit, so that that specific event can be repeated.

Such a wrapper is illustrated in Figure 11. It uses the same parameter sets as the previous parameterized wrapper, plus a new 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 will not be used unless an error is received before the caller has sent an event (which cannot happen in the example connectors shown here).

Figure 13 shows a connector that is somewhat more interesting than the original simple example (in which there were only two events, call and return). 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. Hidden events are used to allow only the server to choose the returned value, and only the client to choose when to close the connection. The final two lines of the Glue are added to model an unreliable implementation of this connector: timeouts may occur when the client is attempting to send requests or to close the connection. Adding these two lines results in a connector that will deadlock, since the client role is not expecting an error. This deadlock will be resolved when we apply the parameterized retry wrapper just seen in Figure 12.

Figure 14 shows the elements of the pattern that must be filled in for the application of the wrapper of Figure 12 to the connector of Figure 13.

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 co-

```
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 example

```
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

operate 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.

5.3 "Failover" and Safety

In the simple hard-wired examples seen earlier, any mistakes in the construction and application of wrappers are likely to be apparent simply by inspection (either of the FSP specification or the corresponding labelled transition system). As the wrappers are generalized and composed, the specification size increases, and this will no longer be the case. However, such defects can still be caught by the safety and progress analyses described earlier.

For example, Figure 15 shows a plausible mistake in the application of a partially parameterized Failover. The event err 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. The mistake will become apparent immediately upon use of 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 error events, as in Figure 3, or ignores error events, as in Figure 7. In this example, the problem would be resolved by removing err from the set CIn, yielding a correct application of the parameterized Failover wrapper.

```
set C = {caller}
set COut = {call}
set CIn = {return, err}
set CErr = {err}
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] → 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.definer:Definer || pri:Glue
      || bk.definer:Backup || bk:BGlue
      || Failover).
```

Figure 15: A Mistake

5.4 In Review

In section 4.2 we began with the goal of being able to take a "faulty" example connector and produce wrappers that will make the connector and definer appear more reliable to the caller. Using our technique it was possible to write wrappers that are easily composable, and to determine via analysis that application of these wrappers to the faulty connector *will* suppress errors and allow the caller to continue to make progress, exactly as desired.

These initial wrappers, however, were hard-wired and could only be used for that particular connector specification. In implementation, they 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 5 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 difficult to apply to connector specifications. Furthermore, even if a mistake is made, it can be readily detected (as shown in 5.3).

In implementation, this generalization into parameterized wrapper patterns, which are then applied to particular connector types, could correspond informally to the use of a tool or "wizard" that is prompted with a wrapper pattern plus a small amount of information about a connector, corresponding to the event-set parameters seen above. 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.). This is the approach that we have used in [19].

6. DISCUSSION

An important consideration when producing a formal specification, or a family of formal specifications (as we have proposed), is the resulting engineering properties of the formal artifact. Our approach has two important properties: *compositionality* and *traceability*. 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 in a section of the model, the corresponding affected section of the implementation can be determined, and vice versa.

As we have demonstrated in the example of section 4.6, this wrapper specification technique exhibits good compositionality. 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 difficult to modify further, and removal of an arbitrary enhancement would not be straightforward.

Traceability between the protocol specification 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 specification, its structure remains similar to the structure of the implementation, enabling the tracing of attributes from a substructure of one to the corresponding substructure of the other. For example, suppose a simple wrapper that correctly implements the description in Figure 11 is applied (inappropriately) to a connector role that initiates more than one kind of event. Analysis of the FSP description of the resulting wrapped connector will show a deadlock that is caused by that particular wrapper; the problem can be traced from there to the implementation of that particular wrapper (more specifically, to its behavior when the timeout exception is caught), even if other wrappers have also been applied to this connector.

Finally, one important question is, can this formalism correspond to implementation as hinted at in section 5.4? 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 [19], 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, including a pattern which re-issues a request when an error is received, and one which adds redundant definer components to a caller/definer communication relationship (as the "failover" wrapper does). 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.

7. CONCLUSION

This work will provide 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 and shown how compositionality is achieved, and how safety and progress analyses may be used to confirm whether a wrapper achieves its intended purpose. Such a technology would allow practitioners to do some reasoning about wrappers' effects in advance of their implementation, as well as to break a complex modification into incremental, more easily understood parts.

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 which also has an increased degree of compositionality, checkability, and traceability.

Several questions were posed in section 3. We have addressed a number of them, showing how to answer them for at least the specific 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, using safety properties, to ensure conformance.
- Compositionality: While confirming idempotence of a wrapper, and checking a pair of inverses, can be addressed in a manner similar to transparency checks, we have not directly addressed other compositional and algebraic properties (such as commutativity).

Future work includes a more generalized analysis of the compositional and algebraic properties of parameterized wrappers, independent of their application to a specific connector specification. For example, given a particular wrapper, we wish to determine whether a particular property such as idempotence will hold no matter what connector it is applied to (provided that the connector meets specific constraints, such as not being subject to deadlock prior to the application of the wrapper).

More work is also needed to complete a set of genericized protocol transformation patterns comparable, and complementary, to the common patterns of connector implementation enhancement described in other work [19]. The examples given here are a step in this direction, giving semantics for a subset of the enhancement patterns that have already been implemented, and thus supporting increased understanding of the results of their application and composition.

8. ACKNOWLEDGMENTS

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

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

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

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

[12] 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.

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

[14] 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.

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

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

[17] M. 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.

[18] 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.

[19] 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.

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