

A Protocol Description Language for Customizing Failure Semantics*

Daniel C. Sturman and Gul A. Agha

Department of Computer Science,
University of Illinois at Urbana-Champaign,
Urbana, IL 61801
{sturman | agha}@cs.uiuc.edu

Abstract

To optimize performance in a fault-tolerant distributed system, it is often necessary to enforce different failure semantics for different components. By choosing a custom set of failure semantics for each component and then by enforcing the semantics with a minimal set of protocols for a particular architecture, performance may be maximized while ensuring the desired system behavior. We have developed DIL, a language for specifying, on a per-component basis, protocols that transparently enforce failure semantics. These protocols may be reused with arbitrary components, allowing the development of a library of protocols.

1 Introduction

Although descriptions of dependability protocols in the literature are relatively simple and concise, incorporating the protocols into an application often requires custom routines which intermix code for the application with that of the protocol. Such intermixing significantly increases the complexity of the code. One way to avoid the resulting complexity is to use a system or language that has been developed to support fault-tolerant computing [5, 13, 11]. However, such support relies on a fixed set of protocols and thereby commits the programmer to defining components with a single *failure semantics* (by failure seman-

tics we mean the expected behavior of an application when a component fails [10]).

A single failure semantics for all components may be satisfactory in some cases. In many systems, however, performance may be improved by having different failure semantics for different components. For example, providing a few low-level but highly dependable servers may save resources. In such a system, the failure semantics of these servers allow fewer failures that are not masked than the failure semantics of the clients. As Christian's has argued, the use of different failure semantics for different components of a system may allow optimization of both performance and dependability [10].

Unfortunately, constructing each component with an optimal set of protocols further complicates the code of distributed applications. Moreover, the failure semantics may change if the system requirements change, or if the program is ported to a new platform. To address these problems, we have developed the Dependability Installation Language, *DIL*, which allows dependability protocols to be developed independently of the applications with which they may be used, *i.e.*, *DIL* allows specification of dependability protocols to transparently enforce a failure semantics.

Protocol specifications in *DIL* are implemented by systems architecture facilities which allows communications between application components to be intercepted; protocols are specified in terms of the communication interface of components. Thus the specification of fault-tolerance is orthogonal to that of a component's functionality.

Protocols may be installed on individual components, supporting the customization of failure semantics. The result is a software development environment where programmers may develop applications without the additional complexity of intermixing code for

*The research described has been made possible by support from the Office of Naval Research (ONR contract numbers N00014-90-J-1899 and N00014-93-1-0273), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program, and by joint support from the Department of Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195).

fault-tolerance. Furthermore, system engineers may independently construct a generic library of protocols for use with a particular platform which may be combined with applications dynamically.

Protocol installation in *DIL* is not limited to a single protocol for each component. Multiple protocols may be specified for a single application to support a complex failure semantics. A collection of *DIL* protocols may be designed where each protocol prevents or masks a particular type of fault. The protocols may then be used together to protect against a specific set of faults. Moreover, because protocols may be reused in different combinations with different applications, it is unnecessary to provide a new implementation for use in every possible combination.

The next section discusses the model of computation used in *DIL*. Section 3 describes the key concepts and abstractions of *DIL*. In Section 4 provides a set of examples to illustrate the the variety of protocols which can be described in *DIL*. As transparency is one of the most important contributions of *DIL*, we discuss how transparency is achieved in Section 5. Section 6 discusses the run-time support necessary for supporting our protocols. The final section provides a detailed comparison of our approach with other systems.

2 Model of Distributed Computing

We develop our approach using the *Actor model* [1, 2]. The model provides an abstract representation of distributed systems. An actor is an encapsulated object that communicates with other actors using asynchronous point-to-point message passing. Specifically, an actor language provides three primitive operators:

send is used to communicate to a given actor. A **send** specifies a destination actor, a method (*i.e.* procedure) to be invoked at the destination, and some values (a message). The message will be buffered in a *mail queue* at the destination until the actor is ready to process it. Each actor has a unique *mail address* which is used to specify a target for communication. Mail addresses may also be communicated in a message, allowing for a dynamic communication topology.

new is used to dynamically create actors. The **new** operator takes an actor behavior (*i.e.* class name) as a parameter, creates a new actor with the behavior and returns its mail address.

become The **become** operator marks the end of state modifications in the execution of a method. Once

a become has executed in a method, the actor may process the next message in its queue.

It is important to note that the concepts described in this paper are not tied to any specific programming language. Our model assumes only that the three actor operators are in some way incorporated in a given language; specifically, we require that new actors may be created dynamically and that the communication topology of a system is reconfigurable.

In fact the actor operators may be used to extend almost any standard sequential language to provide coordination and communication in a distributed environment; local computations may still be expressed in terms of the sequential language. However, note the way in which the sequential and actor constructs are integrated determines the amount of concurrency available in the system.

An actor language may be used as a high level distributed shell to provide interoperability between heterogeneous components. In particular, complex interconnect abstractions can be defined in this language and used to “wrap” existing sequential programs. For example, each method in an actor may invoke a subroutine, or set of routines, written in a sequential language, and dispatch messages based on the values returned. The Carnot project at MCC [18] uses the actor language Rosette in this manner.

In this paper, we integrate the actor operators into an existing language. *Broadway*, the run-time platform we use to implement the ideas presented in this paper, supports *C++* calls for both **send** and **new**; the **become** operator is implicit at the end of each method. Using *Broadway*, developers of distributed programs may use a well known language — *C++* — to develop distributed programs.

3 DIL

In this section we describe the abstractions that are unique to *DIL*; constructs such as timers or failure detectors, which are also common to other languages [14], are explained when they are used in the examples. Protocols in *DIL* consist of a name declaration, a parameter list, an initialization specification, and a set of role definitions. In a protocol definition, a protocol is declared with a name followed by the other information describing the protocol. For example, we declare of our primary-backup protocol as:

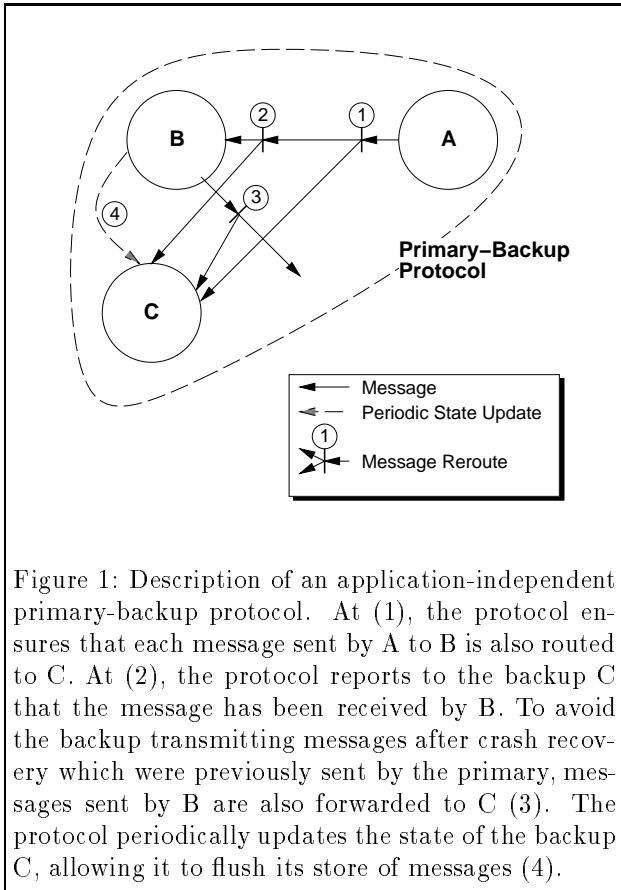


Figure 1: Description of an application-independent primary-backup protocol. At (1), the protocol ensures that each message sent by A to B is also routed to C. At (2), the protocol reports to the backup C that the message has been received by B. To avoid the backup transmitting messages after crash recovery which were previously sent by the primary, messages sent by B are also forwarded to C (3). The protocol periodically updates the state of the backup C, allowing it to flush its store of messages (4).

```

protocol primary-backup {
  Protocol Attributes
  Parameter Declarations
  Initialization
  Role Definitions
}

```

The protocol attributes define how the protocol is installed and how the participants in the protocol interact. Parameters are variables which may be read by all participants in the protocol but may only be set during protocol initialization. Role definitions define how participants in the protocol behave. The different parts of a protocol declaration are explained in greater detail below.

3.1 Roles

DIL protocols are described in terms of the behavior of the different types of participants in the protocol. For a single protocol, each participant is of one of these types termed *roles*. The role of an actor determines how the protocol modifies its behavior to ensure the desired failure semantics.

The actors upon which a protocol is installed has the role of *server*. Protocol installation on an actor

ensures that the actor has the failure semantics that are enforced by the protocol. Actors which are not in the server role are either in an *auxiliary* role or in the role of *client*. An auxiliary role is a role defined by the programmer. For example, the role of *backup* in the protocol shown in Figure 1 is an auxiliary role. The *client* role is the default role: all actors which are not otherwise defined in the protocol, yet which interact with the server, are in the role of client.

Note that roles are with respect to a particular protocol. A single actor may be in a different role for each protocol in which it participates. When two different protocols have been installed on two actors, each actor performs the role of server in its own protocol and the role of client in the other's.

For example, consider a simple example of the primary-backup system shown in Figure 1. This protocol is defined in terms of three roles: server, client, and backup. The actor *A* is in the role of client, the actor *B* is in the role of the server and the actor *C* is in the role of the backup. The protocol details a modification to the actor's behavior for the three roles.

The specification of a role's behavior is referred to as a *role definition*. In *DIL*, a role definition consists of the name of the role in question, a declaration of state, and a set of methods to manipulate the state. The name of the role may be specified by a variable of one of two types: *address* or *group*. An address is a reference to an actor (which may be used to specify the destination of a message). A group is a list of addresses which may also serve as the destination for a message. If the role name is a group, the protocol behavior is applied to all members of the group.

There are two reserved method names which may be used with a role definition: *in* and *out*. These two methods specify the actions taken when a message is sent or received by the actor in the specified role. Messages destined for such an actor must first be processed by the *in* method. Since the difference between a multicast and a unicast message is the type of the destination — *address* versus *group* — multicast operations are treated identically to standard message sends. In this manner, the implementation of the multicast operation may be customized. A message sent by an actor in a role triggers the *out* method. The *in* and *out* methods comprise the heart of protocols based on communication.

Consider the role of client in the primary backup protocol. An actor which interacts with the server, but is not in the role of *backup*, is cast in this role:

```

role client {
    State for the client role.
    method init() {
        Initialization of the state.
    }
    method out(msg m) {
        Multicast the message to the backup and server.
    }
}

```

In this role definition, there are two methods, `init` and `out`. The `init` method is invoked when a new actor assumes the role of *client*. The `out` method is invoked by the run-time system whenever the actor in the role of *client* sends a message. The `out` method multicasts the original message to the server and the backup.

3.2 Protocol Installation

Since protocol names are first-class entities in *DIL*, protocol installation is quite simple. The `install` command installs a protocol on a particular actor. For example, to install the *primary-backup* protocol on an actor *B* (as shown in Figure 1), a programmer may use the command:

```
install(B, primary-backup, <init parameters>);
```

In this manner, *B* may assume the role of *server* in the *primary-backup* protocol. An actor interacting with *B* (such as *A*) may assume the role of *client* in the protocol. The `install` command may take a number of optional arguments to support initialization of the protocol.

Protocols may be installed either symmetrically or asymmetrically. For protocol installed *symmetrically* on a group of actors, each member of the group views the other members as also being in the *server* role in the protocol. For example, in a global checkpoint algorithm installed on the group of actors to be checkpointed, each group member expects the others to exhibit the *server* behavior of performing the checkpointing algorithm.

Conversely, a protocol may be installed *asymmetrically* on a single actor. With asymmetric installation, an actor in the role of *server* never views another actor as in the role of *server*, even if they have the same protocol installed on them. Instead all other actors must be in an auxiliary role or in the role of *client*. In Figure 1, the *primary-backup* protocol has been installed asymmetrically on actor *B*. *B* views *A* as in the *client* role even if *A* also has the asymmetric *primary-backup* protocol installed on it.

How a protocol is installed is specified by the `Installation` attribute in the protocol definition. For example, in the *primary-backup*, which is installed asymmetrically, the programmer may declare:

```
Installation asymmetric;
```

3.3 Isolated Interaction

Some protocols are better expressed with each client-server interaction being isolated from all others. Thus, a protocol may be defined to use *isolated interactions* where a new instance of the role definition is created for each client with which the server interacts. For example, a reliable-messaging protocol may be constructed as a multiple instance protocol where a separate server state controls message delivery information between the server and each client.

Isolated interactions are specified as a protocol attribute. Specifically, for a protocol which uses isolated interactions, attribute is stated as:

```
Isolated-Interaction;
```

3.4 Parameters

To make protocols described in *DIL* more general, it is desirable to be able to specialize a protocol through *parameterization*. In the *primary-backup* example, parameters to the protocol may be the address of the backup object and the time between state updates. Parameters may be of any type. A role definition may be specified for all parameters of type *address* or *group*.

Parameters may only be set in the initialization of the protocol. The initialization has a list of parameters which are used to initialize the protocol parameters. The arguments to the initialization are specified when the protocol is installed on an actor.

Consider the parameter declaration and initialization for the *primary backup* protocol.

```

int timeout;
actor backup;
initialize(int time) {
    timeout = time;
    backup = server.clone();
}

```

Only in the `initialize` routine may the parameters be set. Furthermore, since the parameter `backup` is of type *actor*, it serves as a role in the protocol and may have a role definition.

3.5 Special Protocol Parameters

In addition to user-defined parameters, there are three special parameters: `server`, `client` and `local-client`. The values of these parameters are maintained by the system: they are never set by the user. Furthermore, unlike other parameters, their values may change in context or over time. All three of these parameters are of type `group` or `address`. Thus, each of these three parameters are also a role and may have a role definition.

In each protocol, `server` refers to the object(s) on which the protocol has been installed. The kind of protocol affects the semantics and type of `server`. In symmetric-installation protocols, `server` is of type `group` and specifies the set of objects upon which the protocol has been installed. For asymmetric-installation protocols, `server` is an address which refers to the one object upon which the protocol is installed.

The other two special parameters — `client` and `local-client` — specify default roles for the protocol. In fact, in a system with a dynamic communication topology, all actors that may be involved in communication with a particular server *cannot* be calculated statically. An actor not bound to another role which interacts with a `server` and which resides on the same node as a `server` is in the role of `local-client`. The `client` role refers to all actors not otherwise specified. Although many actors may be in one of these two roles, each of these parameters is of type `actor`: reference to `client` always specifies the actor with which the server is currently interacting.

In the primary-backup example shown in Figure 1, **A** is in the role of `client` or `local-client`, depending on its physical location. If another actor initiated communication with **B** it may assume the role of `client`. Each client, however, may be an instance of the client role description with its own state. **B** is bound to `server`. **C** is not represented by any of the special protocol parameters, instead being represented by the user-defined parameter `backup`.

3.6 Requirements and Null Protocols

A powerful tool used in specifying protocols is *requirements*. A requirement is a statement by one protocol which stipulates the use of a protocol by an actor in an particular role. Programmers must often make assumptions about the failure semantics of other components when designing protocols. These assumptions are expressed as requirements. For example, the

primary-backup protocol may have the requirement:

```
require server is FIFOchannels;
```

This statement may require the server to be also using a protocol that preserves message ordering, possibly forcing the installation of this protocol if unsupported by the hardware. If the underlying hardware supports FIFO ordering, then the `FIFOchannels` protocol should be defined using the `null-protocol` construct.

The `null-protocol` construct defines a protocol without a body, i.e. a protocol that does nothing. Such protocols are used to denote those failure semantics implicitly provided by the underlying hardware. For example, to declare that the network itself guarantees FIFO ordering on messages, the `FIFOchannels` protocol may be declared:

```
null-protocol FIFOchannels;
```

4 Examples

In this section, we provide several examples of protocols constructed using *DIL*. We have chosen three protocols from the literature which are prototypical of the types of protocols which may be specified in *DIL*. Through these three examples, we illustrate how *DIL* may be used to describe a large class of protocols.

4.1 FIFO Channels

We start with a simple example to illustrate the concepts of *DIL*. Specifically, we present a protocol which enforces FIFO ordering of messages delivered to a server. Note that this protocol is similar in structure to checksum protocols, encryption protocols, etc. in that these types of protocols *augment* the message (in this case with a tag) before transmission.

The *DIL* code for our FIFO protocol is shown in Figure 2. In the FIFO protocol, each message is tagged with a unique identifier. Upon receipt of a message, if the tag is not the next in sequence, the message is delayed until all previous messages have been received.

We implement the FIFO protocol using isolated interactions: for each new client communicating with the server, a state for the server is instantiated. There are role definitions for the `local-client`, `client`, and `server`. Furthermore, the `local-client` role definition is empty. This ensures that no operations are performed on actors on the same node as the server: all messages between two local actors are guaranteed to be FIFO

```

protocol FIFO_channel {
  Installation asymmetric;
  Isolated-Interaction;
  role local-client { }
  role client {
    int tag;
    method init() {
      tag = 0;
    }
    method out(msg m) {
      server.tagged_in(tag, m);
      tag = tag + 1;
    }
  }
  role server {
    MsgBag delays;
    int intag;
    method init() {
      intag = 0;
    }
    method tagged_in(int t, msg m) {
      msg next;
      if (t == intag) {
        next = m;
        while (next) {
          deliver next;
          intag = intag + 1;
          next = delays.get(intag);
        }
      } else delays.put(t, m);
    }
  }
}

```

Figure 2: A FIFO-channel protocol.

by the run-time system for which this protocol was designed. If the `local-client` role definition was omitted, however, then the `client` role would apply to *all* clients, regardless of physical location.

4.2 The Primary Backup Protocol

In this section, we describe the *DIL* code for a primary-backup algorithm. This protocol is based on the Auragen Systems protocol described in [7], and we will, therefore, refer to this protocol as the Auragen protocol. The sole difference between the protocol we describe and the Auragen protocol is that the described protocol backs up one actor and does not assume that clients also have a backup: the Auragen protocol assumes all components in the system will have a backup. The Auragen protocol is constructed based on the assumption that the hardware imposes a atomic ordering on multicasts and is different from the example in Section 2. The difference due to architecture is an excellent example of why different protocols may be needed with different hardware. In describing this protocol in *DIL*, we used the original authors'

```

protocol primary-backup {
  Installation asymmetric;
  actor backup;
  int timeout;
  require server is TotalOrderMulticast;
  require client is TotalOrderMulticast;
  initialize(int t)
    Initialize timeout and create backup as clone of server.
  role client {
    method out(msg m)
      Multicast message to both server and backup.
  }
  role server {
    Timer time;
    int incout;
    method init()
      Initialize time and incout.
    method in(msg m, Tag tag)
      Increment incout and deliver message m.
    method out(msg m)
      Multicast message to destination and backup
    method update()
      Send state-update to backup.
      Reset time and incout.
  }
  role backup {
    MsgQueue received;
    int outcount;
    method init()
      Initialize variables.
    method in(msg m)
      If message was sent by server, increment outcount
      Else, record the message in the received queue.
    method update(int incout, State s)
      Reset the backup's state and discard the
      incout oldest messages. Set outcount to 0.
    method failure@server()
      Become the new server. Discard the first
      outcount messages sent by the backup.
  }
}

```

Figure 3: The Auragen primary-backup protocol.

assumptions.

In the Auragen protocol, each message destined for the server is multicast to both the server and the backup. The server processes these messages but the backup simply logs their arrival. When the server sends a message, it is multicast to both the destination and its backup. The backup logs the transmission of the message. The server periodically sends a state update to the backup and the number of messages processed since the last state update. The backup then updates its state and discards the appropriate logged messages.

We define the Auragen protocol to be installed asymmetrically: unlike with symmetric installation protocols, two servers which have primary-backup installed on them will view each other in the role of client. The *DIL* code for the protocol is shown in Figure 3. To keep the example to a reasonable size, the methods descriptions have been replaced with pseudo-code.

Our primary-backup protocol is described in terms of two parameters, two requirements, and three role definitions. The `backup` parameter is initialized as the address of a clone of the server actor. The other parameter, `timeout`, is the amount of time between state updates of the backup by the server. The assumptions made for this protocol (i.e. that all multicasts are totally ordered) are expressed in terms of the two *requirements*. In the case of the Auragen hardware, `TotalOrderMulticast` would be defined as a null-protocol: the protocol is supplied by the underlying hardware.

The three role definitions detail the behavior of the protocol at the clients, the server, and the backup. Because no definition was given for the `local-client` role, the `client` role will be applied to all actors which are not the server or the backup, regardless of physical location. Whenever an actor attempts to communicate with an actor upon which this protocol has been installed (i.e. a server), the `client` role definition will be applied to it.

The method `failure@server` is defined to be invoked upon failure of the server. Any actor may be watched for failure in *DIL* by naming a method `failure@<address>`. In this case, the backup watches for a failure at the server.

4.3 Distributed Checkpoint

The *DIL* specification of our distributed checkpoint algorithm is based on the one presented in [9]. When the protocol runs, it results in a distributed snapshot of the system being saved to stable storage. The system may later be restored from one of these checkpoints if a failure occurs. The *DIL* code for this example is shown in Figure 4.

We use this example to illustrate the effectiveness of symmetric installation protocols. As mentioned in Section 2, the parameter `server` is of type *group*. Thus, each actor upon which the protocol is installed becomes a member of this group. All actors using the `DistributedCheckPoint` algorithm will be a member of the `server` group. Furthermore, the use of the `require` statement ensures that any actor which communicates with an actor in the group joins the group itself: a `require` statement enforces the immediate installation of

```

protocol DistributedCheckPoint {
  Installation asymmetric;
  actor starter;
  actor manager;
  require client is DistributedCheckPoint;
  require server is FIFOChannels;
  initialize(group root, int period) {
    starter = new Start(root,period);
    manager = new NeighborManager(root);
  }
  role server {
    group neighbors;
    Tally received;
    actor storefile;
    boolean checkpointing;
    QueueList channels;
    method init()
      Contact manager to setup neighbors. Open storefile.
    method start_cp(actor from) {
      if (!checkpointing) {
        checkpointing = true;
        storefile.write(self.getstate());
        neighbors.start_cp(self);
      }
      received.mark(from);
      storefile.write(channels.at(from));
      if (received.full()) {
        checkpointing = false;
        Reset all variables
      }
    }
    method in(msg m) {
      if (checkpointing && (!received.marked(m.src)))
        (channels.lookup(m.src)).put(m);
      deliver m;
    }
  }
}

```

Figure 4: A distributed checkpoint protocol.

the checkpoint algorithm on all clients. The other `require` statement enforces the use of the `FIFOChannels` protocol by all members in the group.

5 Transparency

As shown above, each *DIL* protocol is described in terms of operations on generic messages. Thus, each *DIL* protocol is a description of how the communication behavior of a group of actors must be modified to enforce the desired failure semantics. Since these operations are completely independent of any application upon which the protocol may be installed, *DIL* protocols may be used transparently with any group of actors. Thus, our implementation is designed to support per-actor transparency. Specifically, our system support must allow:

- Dynamic installation of a communication behavior implementing a protocol. Dynamic installation allows new clients to adapt new protocols as they interact with new servers. Furthermore, additional protocols may be imposed on servers while the system is running.
- Composition of these modifications. Composition is necessary to allow multiple protocols to be used with a single actor. The (impractical) alternative is to have a single modification for each possible combination of protocols.

We employ *reflection* [16] as the enabling technology to support dynamic modification of an actor’s communication behavior. Reflection means that an application can access and manipulate a description of its own behavior and execution environment. In a sense, the modification of this description *reflects* onto the described actor. These descriptions are themselves actors which may interact with other actors by sending and receiving messages. For a single actor, the actors representing such a description comprise the *meta-level* of that actor; the actors being described are termed the *base-level* actors. Since the meta-level actors are actors and are accessible in the running system, reflective modifications may be made dynamically.

The most general form of reflection requires that meta-level actors be interpreters supporting any possible change to their base-level actors. Such an approach is inefficient and unsafe. For our purposes, we use a limited reflective model in which only the communication behavior of application components may be customized. This model may be implemented efficiently and safely since it consists only of compiled actors which are bound at run-time to a single actor by a pointer. Our meta-level architecture MAUD¹ allows the customization of the transmission and reception behavior of a component in this manner [3, 4].

Specifically, the behavior is modified in terms of a communication meta-level actor which provides a representation of both the dispatch and reception behaviors of an actor. This actor, called a *communicator*, encapsulates a state, has a set of methods, and may send and receive messages as with any actor in the system.

The communicator’s `transmit` method defines the transmission behavior of the base actor. All messages sent by the base-level actor are rerouted to the communicator as the argument to a `transmit` method. By

¹MAUD stands for **M**eta-level **A**rchitecture for **U**ltra **D**ependability

installing a communicator on an actor, the base-level actor’s dispatch behavior may be modified.

The reception behavior of an actor may also be modified. Each communicator also serves as the mail queue for its base-level actor. Thus the meta-actor, rather than the system default, serves as the mail buffer holding messages received by the component. The customized mail queue is then accessed by the the run-time system using the `get` method to retrieve the next message and the `put` method to deliver a new message. The use of the customized mail queue in our implementation is hidden from the base-actor by the run-time system: an actor sees no difference between the system-default mail queue and one provided by a communicator.

Given the ability to modify an actor’s communication behavior through reflection, we also gain dynamic installation and composition of protocols. Since we bind a communicator to a base-actor through the use of a pointer, this pointer may be modified dynamically to change the meta-level actor in use.

Composition of dependability protocols is achieved by transparently manipulating the meta-level of the meta-level. Since communicators are themselves actors in their own right, their communication behavior may also be modified. Consider an actor *C* with communicator *B*. *B*, in turn, has the communicator *A*. The communicator *A* modifies the communication behavior of *B*. In this manner, the protocol represented by *A* is performed first and the protocol represented by *B*, second. To compose two protocols, we install the second protocol on *B* and, thus, *C* is affected by both protocols.

6 Implementation

We are currently developing a *DIL* compiler and incorporating the necessary run-time support into the *Broadway* platform. *Broadway* supports actors written in *C++*, providing actor scheduling, creation, communication, I/O, and migration. *Broadway* also implements the reflective architecture MAUD [17] to support customization of an actor’s communication behavior as described in Section 5.

We use MAUD to provide the underlying implementation machinery for our protocol. The *DIL* compiler translates *DIL* protocols into a set of actors. For each role definition in the protocol, a specialized communicator class is created. This class implements the behavior described for the role. For each protocol, a protocol manager class is created which has methods

for the customization of a new actor with the appropriate meta-actor. These methods handle the installation of the protocol at a new server, the first interactions with a new client, and the initialization of the protocol including the initialization of protocol parameters. Work on the compiler is currently in progress.

There is minimal expense involved in using MAUD to customize objects. As mentioned in Section 5, a communicator is bound to an object via a pointer. The system then knows to route all of the object's outgoing messages to the communicator and to first deliver all incoming messages to the communicator. Communicators are either local to the object being customized or are remote due to the nature of the protocol itself. Therefore, the indirection is minimal when compared to the general cost of communication in a distributed system.

In addition to MAUD, additional run-time support will be needed to support dynamic installation of protocols and the correct use of protocols by clients. The protocols needed to enforce the failure-semantic of the run-time system may vary between platforms. We support varying failure semantics for the system by implementing the modules for protocol control as actors. Protocols may then be installed on these components. The protocols to be used with these components will be specified for each node in a system initialization script.

7 Discussion

In this paper, we have presented the language *DIL* for use in describing fault-tolerance protocols. Using *DIL*, protocols may be installed on individual actors, allowing the specification and implementation of failure semantics on a per-component basis. Construction is simplified by protocols which may be composed without adversely affecting each other. By describing protocols orthogonally from applications, application development may be simplified and reusable libraries of protocols may be developed for use with different hardware platforms.

Although designed to describe fault-tolerance protocols, *DIL* should not be confused with other protocol description languages such as LOTOS [6]. These languages do not address transparency or roles in a protocol. Instead, their use lies in expressing the concurrent behavior involved in communication protocols. Thus, this work is orthogonal to the goals and design of *DIL*.

Customization is not unique to *DIL*. Systems such as *Choices* [8] or the x-Kernel [12] support the cus-

tomization of the operating system. However, operating system customization does not allow the per-object specification of failure semantics. Other reflective language systems such as ABCL/R [21] or the Muse Operating System [20] do allow the customization of objects. However, none of these systems are aimed towards fault-tolerant computing. For our purposes, they are too general, in some cases requiring reinterpretation of an object's behavior.

The work done by Schlichting and Thomas with SR-FT [15] is closest in approach to ours. The language allows the construction of objects with stronger failure semantics from objects with less stringent ones. However, SR-FT relies on the general concept of the *fail-stop object* which limits the type of failure semantics which may be enforced. For example, SR-FT would not be suitable for expressing the FIFO ordering protocol or the distributed checkpoint protocol presented in Section 4.

There are some concepts, however, that *DIL* does not describe well. For example, transaction systems are highly dependent on modification of an application's code. The items comprising a transaction and the ability to handle a transaction abort must all be specified by the application programmer and, therefore, cannot be specified independently via a *DIL* protocol. Even in these systems, *DIL* may be used to describe the most efficient commit or recovery strategy for a given transaction. Moreover, if special *DIL* methods similar to *out* or *in* were added to signal the beginning and end of a transaction, transaction programs could then be written with the optimal transaction control protocols for a particular platform being expressed in *DIL*.

Another area which *DIL* does not currently support is real-time constraints. Through the use of timers, a *DIL* protocol may be constructed to signal an exception if a task does not finish within a current time bound. However, *DIL* does not currently present constructs or an interface to the scheduler for specifying real-time constraints.

Despite these limitations, we believe the constructs of *DIL* allow a large class of protocols to be easily described. In the future, we will develop tools for reasoning about protocols. Initial work on reasoning about a meta-level architecture for communication is described in [19]. We are continuing our work on the programming environment for *DIL* and are developing tools to simplify the composition of protocols and applications.

Acknowledgments

We would like to thank Svend Frølund, Wooyoung Kim, Rajendra Panwar, and Shangping Ren for discussions concerning the development of *DIL* and for reading previous versions of this paper.

References

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [2] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.
- [3] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A Linguistic Framework for the Dynamic Composition of Dependability Protocols. In C.E. Landwehr, B. Randell, and L. Simoncini, editors, *Dependable Computing for Critical Applications 3*, volume VIII of *Dependable Computing and Fault-Tolerant Systems*, pages 345–363. IFIP Transactions, Springer-Verlag, 1993.
- [4] Gul Agha and Daniel C. Sturman. A Methodology for Adapting to Patterns of Faults. In Gary Koob, editor, *Models and Frameworks for Dependable Systems*, volume I of *Foundations of Dependable Computing*, chapter 1.2. Kluwer Academic Publishers, 1994.
- [5] K. P. Birman and T. A. Joseph. Communication Support for Reliable Distributed Computing. In *Fault-tolerant Distributed Computing*. Springer-Verlag, 1987.
- [6] Tommaso Bolognesi and Ed Brinksmas. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, (14):25–59, 1987.
- [7] Anita Borg, Jim Baumbach, and Sam Glazer. A Message System Supporting Fault-Tolerance. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 90–99, 1983.
- [8] Roy Campbell, Nayeem Islam, David Raila, and Peter Madany. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*, pages 117–126, September 1993.
- [9] K. Mami Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
- [10] Flaviu Cristian. Understanding Fault-tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, 1991.
- [11] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *CAMELOT AND AVALON: A Distributed Transaction Facility*. Morgan Kaufmann Publishers, Inc., 1991.
- [12] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–75, January 1991.
- [13] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.
- [14] Richard D. Schlichting, Flaviu Christian, and Titus D. M. Purdin. A Linguistic Approach to Failure Handling in Distributed Systems. In A. Avižienis and J.C. Laprie, editors, *Dependable Computing for Critical Applications*, pages 387–409. IFIP, Springer-Verlag, 1991.
- [15] Richard D. Schlichting and Vicraj T. Thomas. A Multi-Paradigm Programming Language for Constructing Fault-Tolerant, Distributed Systems. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 222–229, 1992.
- [16] B. C. Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, Massachusetts Institute of Technology. Laboratory for Computer Science, 1982.
- [17] Daniel C. Sturman. Fault-Adaptation for Systems in Unpredictable Environments. Master’s thesis, University of Illinois at Urbana-Champaign, January 1994.
- [18] C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk. The Extensible Services Switch in Carnot. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2), May 1993.
- [19] Nalini Venkatasubramanian and Carolyn Talcott. A MetaArchitecture for Distributed Resource Management. In *Proceedings of the Hawaii International Conference on System Sciences*. IEEE Computer Society Press, January 1993.
- [20] Y. Yokote, A. Mitsuzawa, N. Fujinami, and M. Tokoro. The Muse Object Architecture: A New Operating System Structuring Concept. Technical Report SCSL-TR-91-002, Sony Computer Science Laboratory Inc., February 1991.
- [21] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*, chapter Reflection in an Object-Oriented Concurrent Language, pages 45–70. MIT Press, Cambridge, Mass., 1990.