To appear in: G. Koob (ed.), *Foundations of Ultradependability, Vol. 1,* Kluwer Academic 1994

SECTION 1.2

A Methodology for Adapting to Patterns of Faults*

Gul Agha and Daniel C. Sturman †

Abstract

In this paper, we present a language framework for describing dependable systems. Our framework emphasizes *modularity* and *composition*. The dependability and functionality aspects of an application may be described independently, providing separation of design concerns. Furthermore, the dependability protocols of an application may be constructed bottom-up as simple protocols that are composed into more complex protocols. Composition makes it easier to reason about the behavior of complex protocols and supports the construction of generic reusable dependability schemes. A significant aspect of our language framework is that dependability protocols may be loaded into a running application and installed dynamically. Dynamic installation makes it possible to impose additional dependability protocols on a server as clients with new dependability demands are integrated into a system. Similarly, if a given dependability protocol is only necessary during some particular phase of execution it may be installed during that period only.

1.2.1 Introduction

A number of systems have been developed to support the development of dependable computing applications. Such support is given in terms of *failure semantics* which specify legal ways in which a component can fail [11]. Failure semantics are enforced through the use of *dependability protocols* which guarantee that the probability of a failure of a type not specified in the semantics is acceptably small. However, existing systems assume that the *failure semantics* of a service are static and, therefore, the dependability protocols used may be fixed.

In many computer systems, it is either unsatisfactory to adhere to a static failure semantics or impossible to adequately enforce the semantics with a fixed group of dependability protocols. We illustrate this concept with two example systems:

- Consider an embedded system which is required to function over a long duration, yet is fault-prone due to the uncertain environment in which it operates. If this system is physically isolated, such as in the control system of a satellite, physical modification of system components

*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). The research described in here has benefitted from fruitful discussions with, and critical comments from, Christian Callsen, Svend Frólund, WooYoung Kim, Rajendra Panwar, Anna Patterson, Shangping Ren, Carolyn Talcott, Nalini Venkatasubramaniam, Takuo Watanabe among others.

†Authors address: Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue, Urbana, Illinois 61801, USA. Email: { agha | sturman }@cs.uiuc.edu

is often infeasible. In such a system, a change in the physical environment may result in protocols designed for the old environment failing to uphold the failure semantics in the new environment. A different group of dependability protocols may then be required to enforce the desired failure semantics of the system.

- Consider an *open system*. Open systems allow interactions with the external environment; in particular, new services may be added or deleted dynamically from the system in response to external events. Consequently, it may not be possible to statically determine the system configuration. Without knowing the system configuration, it may not be possible to determine what failure semantics a process must have, or what protocols are necessary to enforce these semantics, until after the process actually joins the system. Furthermore, the addition of new services may require a change in the failure semantics of existing components. For example, a file server may initially address safety only by check-pointing the files to stable storage. New clients that are added to the system, however, may require the server to also provide *persistence* and a protocol to support replication may need to be added.

In this paper, we describe a methodology for the modular specification of systems that adapt to patterns of faults. We call the resulting systems *adaptively dependable*. We present a methodology which allows the transparent installation and reuse of dependability protocols as well as their dynamic installation in a system. Our methodology, when combined with a suitably structured exception handling mechanism and fault detection, allows for the development of fault handlers which can maintain consistent failure semantics within a changing environment and can alter failure semantics as system needs change. We have provided programmer support for our methodology in the language *Screed* which is implemented on our run-time system *Broadway*.

We employ *reflection* as the enabling technology for dynamic installation of dependability protocols. Reflection means that an application can reason about and manipulate a representation of its own behavior. This representation is called the application's *meta-level*. The components of an object that may be customized at the meta-level are referred to as the *meta-architecture*. In our case, the meta-level contains a description which implements the failure semantics of an executing application; reflection thus allows dynamic changes in the execution of an application with respect to dependability.

Besides supporting dynamic installation, our meta-architecture supports transparency and reuse of dependability protocols. For example, the meta-architecture allows protocols to be expressed as abstract operations on messages. Since the individual fields of a particular message are never examined, the same protocol may be used with different applications.

Given this technique for dynamic modification of the dependability protocols used in a system, we describe how fault-detection and exception handling may be used in conjunction with our meta-architecture to support adaptive dependability. We model both failures and exceptions as objects. Each type of fault which may be detected is described as a specific system exception.

We construct managers — exception handlers with meta-level capabilities — to address system exceptions. Managers serve three purposes:

- A manager may correct for recoverable faults. The corrections allow the system to continue to function despite a fault. This role is generally referred to as performing forward error recovery.
- Managers provide failure prevention. When a manager discovers a pattern of component failures, it dynamically installs protocols which *mask* future failures or facilitate future fault-correction by expanding the set of recoverable faults. In this manner, we have taken forward error recovery one step further: rather than simply adjusting the system state, the actual dependability characteristics of the system may be modified.

- Managers support reconfiguration of the dependability protocols in a system. This may be done either to alter the system's failure semantics or to correctly enforce these semantics once the environment changes. Thus, we can develop dependable long duration systems whose fault patterns are not known at start-up time.

A prototype implementation of a run-time system which tests these ideas is described: the system *Broadway* supports our meta-architecture as well as failure detection and a set of system exceptions. On top of Broadway, we have implemented the language *Screed*. Screed is a prototype concurrent actor language which provides complementary constructs for both fault-detection through exception handling and dynamic installation of protocols through a meta-architecture. Screed is presented as a demonstration of how such constructs may be added to existing languages.

This paper is organized as follows. Section 1.2.2, discusses related research in the areas of reflection, exception handling, and languages for fault-tolerance. Section 1.2.3 provides a brief description of the concepts of reflection, the Actor model, and object-orientation. Section 1.2.4 provides a guide to the syntax of Screed to assist in understanding our examples. Section 1.2.5 discusses our meta-level architecture and how it may be used to construct dependability protocols. We also discuss the effect of our meta-level architecture on protocol performance. Section 1.2.6 describes exception handling in Screed and how exception handling may be used in conjunction with our meta-level architecture to implement adaptively dependable systems. We then illustrate this technique with an example of a system adapting to a change in environment. In

1.2.2 Related Work

A number of languages and systems offer support for constructing fault tolerant systems. In Argus [23], Avalon [15] and Arjuna [31], the concept of nested transactions is used to structure distributed systems. Consistency and resilience is ensured by atomic actions whose effects are check-pointed at commit time. The focus in [27], [9] and [7] is to provide a set of protocols that represent common communication patterns found in fault tolerant systems. None of the above systems support the factorization of fault tolerance characteristics from the application specific code. In [38] and [28], replication can be described separate from the service being replicated. Our approach is more flexible: fault tolerance schemes may not only be described separately, they may be attached and detached dynamically. Another unique aspect of our approach is that different fault tolerance schemes may be composed in a modular fashion. For example, check-pointing may be composed with replication without requiring that the representation of either protocol know about the other.

Non-reflective systems which support customization do so only in a system-wide basis. For example, customization in a micro-kernel based system [1] affects all the objects collectively. In an object-oriented system such as *Choices* [8], frameworks may be customized for a particular application. However, once customized, the characteristics may not change dynamically. Reflection in an object based system allows customization of the underlying system independently for each object. Because different protocols are generally required for very specific subsets of the objects in a system, this flexibility is required for implementing dependability protocols.

Reflection has been used to address a number of issues in concurrent systems. For example, the scheduling problem of the Time Warp algorithm for parallel discrete event simulation is modeled by means of reflection in [40]. A reflective implementation of object migration is reported in [37]. Reflection has been used in the Muse Operating System [39] for dynamically modifying the system behavior. Reflective frameworks for the Actor languages MERING IV and Rosette have been proposed in [16] and [35], respectively. In MERING IV, programs may access *meta-instances* to modify an object or *meta-classes* to change a class definition. In Rosette, the meta-level is described in

terms of three components: a *container*, which represents the acquaintances and script; a *processor*, which acts as the scheduler for the actor; and a *mailbox*, which handles message reception

The concept of unifying exception handling and fault detection was originally proposed in [30] and then refined in [29]. In these papers, detected failures are considered asynchronous events much as exceptional conditions are treated in distributed programming languages. Therefore, exception handling construct provide a natural way to incorporate failure-response code into an application.

Goodenough introduced the idea of exceptions and exception handling in [19]. Since then, many different exception handling mechanisms have been proposed. Exception handling constructs have been developed for object-based languages such as Clu [24] and Ada [12]. Dony [13] describes an approach for object-oriented languages and its implementation in Smalltalk. In this approach, exceptions are implemented as objects much as we do. Exception handling for C++ is discussed in [33]. A good overview of techniques proposed for other object-oriented languages can be found in [14].

A critical difference between object-oriented approaches to exception handling and non-object-oriented approaches such as CLU [24] or Ada [12] is that, in the latter, the exception object is represented by a set of parameters to a function. Therefore, on generating the signal, a parameter list must provide all possible information used by the handler.

For concurrent systems, another technique has been proposed for languages which use RPC communication [10]: the technique is based on synchronized components which allows the exception handling constructions to be closer to that of a sequential system than an asynchronous system.

Exception handling mechanisms have been proposed for other Actor languages. An exception handling mechanism was proposed for ABCL/1 and for Acore [21, 26]: the mechanism uses *complaint addresses* to support exception handling. A complaint address is a specific location, specified with each message, to which all signals are dispatched.

1.2.3 Background

Before discussing our meta-architecture and how we use it to support adaptive dependability, we first discuss in greater detail some concepts that are important in our framework. The organization of this section is as follows. First, we briefly discuss some of the advantages of object-oriented programming and how they are useful with our methodology. Secondly, we describe the Actor model of concurrent computation. We chose the Actor model as the basis of our work due to the ease with which it may be extended, Finally, we give a more in-depth discussion of reflection and how it relates to a programming language.

Object Orientation

In an object-oriented language, a program is organized as a collection of objects. Each object is an encapsulated entity, representing an instance of an abstract data type. The local data comprising each object may only be accessed through an interface specified as a set of *methods*. The operations carried out by a method are not visible outside the object. Objects communicate with messages which invoke a method in the receiving object. The local data of another object cannot otherwise be accessed or modified.

Objects are *instantiated* from *classes*. A class is a user-defined abstraction. Classes may be thought of as types and objects as elements of that type. Instantiation is the creation of an object of a particular class. Classes contain the description (code) of the methods and of the instance variables for objects instantiated from that class. Classes may *inherit* from other classes. Inheritance provides the inheriting class with the properties – the methods and instances – of the *ancestor* class. The

inheriting class can then utilize these properties as well as augment them with new instances variables or methods. Methods may be inherited directly or redefined, facilitating code reuse.

Object-oriented languages allow for modular development of systems. The implementation of each component is hidden from other components: only the interface is known. In this way, a component's implementation may change without affecting other components. Code may also be reused efficiently since components may share code by inheriting from a common ancestor class. Note that our use of classes and inheritance differs from that in sequential object-oriented languages in that we do not support class variables.

The Actor Model

We illustrate our approach using the *Actor model* [2, 3]. Actors can be thought of as an abstract representation for multicomputer architectures. An actor is an encapsulated object that communicates with other actors through asynchronous point-to-point message passing. Specifically, an actor language supports three primitive operators:

send Actors communicate through asynchronous, point-to-point message passing. The **send** operator is used to communicate a message asynchronously to another actor. Each message invokes a method (or procedure) at the destination. Upon reception of the message at the destination, the message will be buffered in a *mail queue*. 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 Actors may dynamically create other actors. The **new** operator takes an actor behavior (class name) as a parameter, creates a new actor with the correct behavior and returns its mail address. The mail address is initially known only by the creating actor. However, the creator subsequently include this new mail address in future messages.

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 continue to modify state local to the method. However, such state changes do not effect the way in which the actor may process the next message. Therefore, once this operator is executed, the actor may begin processing its next pending message. Judicious use of the **become** operator may improve performance by allowing *internal* concurrency: i.e., multiple threads of execution within a single actor.

It is important to note that the idea of using reflection to describe dependability is not tied to any specific programming language. Our methodology assumes only that these three operators are in some way incorporated into the language; 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. The level at which the sequential and actor constructs are integrated determines the amount of concurrency available in the system.

An actor language may be used to "wrap" existing sequential programs, serving as an interconnection language. With this approach, each method in an actor class invokes a subroutine, or set of routines, written in a sequential language and dispatches messages based on the values returned. Such an approach was taken by the Carnot project at MCC [34]. In Carnot, the actor language Rosette "glues" sequential components together to facilitate heterogeneous distributed computing.

A complementary approach is to actually integrate the actor operators into an existing language. *Broadway*, the run-time platform we use to implement the ideas 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.

Actor operators have also been combined with functional languages. Specifically, actor operators have been added to the call-by-value λ-calculus [5]. In this case, the local computation is modeled as a sequential functional computation. An operation semantics is developed for the resulting language. The semantics supports operational reasoning. In [36], the semantics is extended to support formal reasoning about meta-architectures such as the one we describe here.

If necessary, the actor operators may also be extended to support more complex functionality. In particular, communication model may be modified to support more complex message passing constructs. The asynchronous point-to-point communication model for actors has been extended to include pattern-based multicasts using *ActorSpaces* [6]. Furthermore, remote procedure calls may be transformed into a set of asynchronous messages using a concurrent analog of the continuation passing style [22].

Synchronization constraints [17] and multi-object constraints [18] are two other extensions of the actor operators which greatly simplify distributed programming. Constraints allow the programmer to specify "when" asynchronous events may occur based on the state of a single object or the occurrence of other events in the system. Using these techniques, the non-determinism of asynchronous communication may be constrained to maintain a consistent system state without requiring an overly restrictive communication model.

Reflection

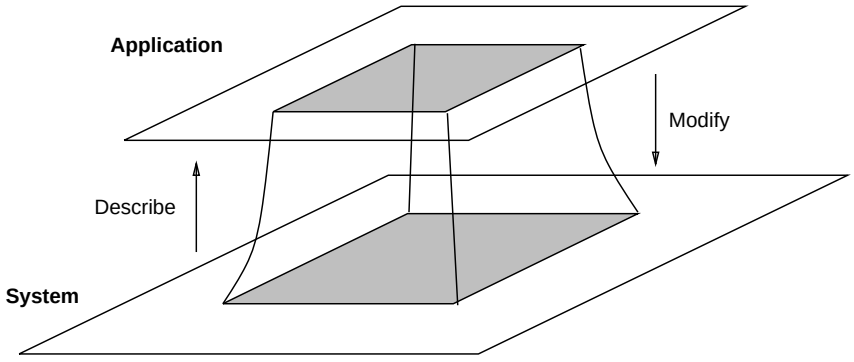

Figure 1.2.1: Through reflection, an application may modify the system by modifying its *meta-objects*. Meta-objects are a system level description of the base-level application objects.

Reflection means that a system can manipulate a causally connected description of itself [32, 25]. Causal connection implies that changes to the description have an immediate effect on the described object. In a reflective system, a change in these descriptions or *meta-objects* results in a change in how objects are implemented. The object for which a meta-object represents certain aspects of the implementation is called the *base object*. This relationship is shown in Figure 1.2.1.

Meta-objects may be thought of as objects which logically belong in the underlying run-time system. For examples, a meta-object might control the message lookup scheme that maps incoming

messages to operations in the base object. Another meta-object may modify how values are read from memory. Using reflection, such implementation level objects can be accessed and examined, and user defined meta-objects may be installed, yielding a potentially customizable run-time system within a single language framework.

The reflective capabilities which are provided by a language are referred to as the *meta-level architecture* of the language. The meta-level architecture may provide variable levels of sophistication, depending on the desirable level of customization. The most general meta-level architecture is comprised of complete interpreters, thus allowing customization of all aspects of the implementation of objects. In practice, this generality is not always needed and, furthermore, defining a more restrictive meta-level architecture may allow reflection to be realized in a compiled language. The choice of a meta-level architecture is part of the language design. Customizability of a language implementation must be anticipated when designing the run-time structure. Although a restrictive meta-level architecture limits flexibility, it provides greater safety and structure. If all aspects of the implementation were mutable, an entirely new semantics for the language could be defined at run-time; in this case, reasoning about the behavior of a program would be difficult.

We limit our meta-level to contain only the aspects that are relevant to dependability. Application specific functionality is described in the form of base objects and dependability protocols are described in terms of meta-objects. Thus, dependability is modeled as a special way of implementing the application in question. Our methodology gives modularity since functionality and dependability are described in separate objects. Since meta-objects can be defined and installed dynamically, the objects in a system can dynamically change the protocols enforcing their failure semantics as system needs change. Furthermore, new dependability protocols may be defined while a system is running and put into effect without stopping and recompiling the system. For example, if a communication line within a system shows potential for unacceptable error rates, more dependable communication protocols may be installed without stopping and recompiling the entire system.

Since meta-objects are themselves objects, they can also have meta-objects associated with them, giving customizable implementation of meta-objects. In this way, meta-objects realizing a given dependability protocol may again be subject to another dependability protocol. This scenario implies a hierarchy of meta-objects where each meta-object contributes a part of the dependability characteristics for the application in question. Each meta-object may be defined separately and composed with other meta-objects in a layered structure supporting reuse and incremental construction of dependability protocols.

Because installation of a malfunctioning meta-level may compromise the dependability of a system, precautions must be taken to protect against erroneous or malicious meta-objects. To provide the needed protection of the meta-level, we introduce the concept of privileged objects called *managers*. Only managers may install meta-objects. Using operating system terminology, a manager should be thought of as a privileged process which can dynamically load new modules (meta-objects) into the kernel (meta-level). It should be observed that, because of the close resemblance to the operating system world, many of the operating system protection strategies can be reused in our design. We will not discuss particular mechanisms for enforcing the protection provided by the managers in greater detail here. Because only managers may install meta-objects, special requirements can be enforced by the managers on the structure of objects which may be installed as meta-objects. For example, managers may only allow installation of meta-objects instantiated from special verified and trusted libraries. Greater or fewer restrictions may be imposed on the meta-level, depending on the dependability and security requirements that a given application must meet.

1.2.4 Screed

Screed is an object-oriented actor language that compiles applications for Broadway. Screed will be used to illustrate examples in this paper, Screed is an object-oriented language: programs are written in terms of class definitions. Each class defines a single actor behavior and consists of a set of variable declarations and a set of method definitions. Screed supports inheritance. A class for which a parent class is not specified will, by default, inherit from the system defined `Object` class. At any point, a parent method may be referenced by sending a message to the "object" `parent`. Inheritance is specified when the class is defined:

```
class MyMailQueue : MailQueue {
    ... instance variables ...
    get() {
        ... method body...
    }
    put() {
        ... method body...
    }
}
```

In this example, the class `MyMailQueue` with the methods `get` and `put` is defined. It inherits from the class `MailQueue`.

Classes may be instantiated using the `new` command which returns a new actor address. For example:

```
foo = new MyMailQueue;
```

This statement creates a new actor with the behavior `MyMailQueue` and returns the address of this actor, which is assigned to `foo`.

There are five primitive types in Screed. The types `int`, `real`, and `string` are self-explanatory. The type `actor` holds the address of any actor, regardless of class. The type `method` can have the value of any legal method name. In addition, one-dimentional arrays of any of these types may be specified. Arrays are defined and used as in C++:

```
actor myReplicas[5];
   ⋮
myReplicas[2] = new ...;
```

Actors communicate through asynchronous message passing. In the current implementation of Broadway message ordering (from a given source to the same destination) is guaranteed, although actor semantics enforce no such requirement. Messages are sent by specifying a method and an actor address with the appropriate parameters:

```
foo.get();
```

In this case, the method `get` is invoked on the actor `foo` without any parameters. Since methods are first-class values, it would be possible to specify a variable instead of the name of a particular method. Parameters are dynamically type-checked upon reception at the message destination. Note that since we are using asynchronous message passing, this method invocation does not block.

Although asynchronous message passing provides improved performance and concurrency, a drawback is the difficulty in providing return values: since the method does not block upon sending a message, it is necessary to specify a return address and method in the message itself. Therefore, method invocations may return a value, thereby acting as an remote procedure call (rpc). For example:

```
A
x = foo.get();
B
```

With rpc-communication, the current method invocation will block. The instructions in A will execute, followed by a message send to `foo`. B will not execute until a return value arrives and the value is assigned to `x`.

In a asynchronous system, the programmer may want to prevent certain methods from executing based on an actor's state. Therefore, we support *synchronization constraints* [17] in Screed. Using synchronization constraints, the programmer will be able to specify exactly which methods may not be invoked. Maximal concurrency is then preserved since only the minimal synchronization — as specified by the programmer not the language – will be enforced.

The other constructs which comprise expressions in Screed (`if`, `while`, etc.) are similar to those in C; we do not describe them further.

1.2.5 Meta-level Architecture for Ultra-dependability

In this section we introduce MAUD (**M**eta-level **A**rchitecture for **U**ltra **D**ependability) [4]. MAUD supports the development of reusable dependability protocols. These protocols may then be installed during the execution of an application. MAUD has been implemented on Broadway, our run-time environment for actors.

We begin with a discussion of MAUD's structure. We then discuss how transparency and reusability of protocols are supported by MAUDand provide an example to illustrate the concepts. We finish this section by demonstrating how MAUD also allows the composition of protocols and give an example of composition.

A Meta-Level Architecture

As previously mentioned, MAUD is designed to support the structures that are necessary to implement dependability. In MAUD, there are three meta-objects for each actor: *dispatcher*, *mail queue* and *acquaintances*. In the next three paragraphs we describe the structure of meta-objects in MAUD. Note that MAUD is a particular system developed for use with actors. It may be possible, however, to develop similar systems for other models.

The *dispatcher* and *mail queue* meta-objects customize the communication primitives of objects so that their interaction can be modified for a variety of dependability characteristics. The dispatcher meta-object is a representation of the implementation of the message-send action. Whenever the base object issues a message send, the run-time system calls the `transmit` method on the installed dispatcher. The dispatcher performs whatever actions are needed to send the given message. Installing dispatchers to modify the send behavior makes it possible to implement customized message delivery patterns.

A *mail queue* meta-object represents the mail queue holding the incoming messages sent to an actor. A mail queue is an object with `get` and `put` operations. After installation of a mail queue meta-object, its `get` operation is called by the run-time system whenever the base object is ready to process a message. The `put` operation on a mail queue is called by the run-time system whenever a message for the base object arrives. By installing a mail queue at the meta-level, it is possible to customize the way messages flow into the base object.

The *acquaintances* meta-object is a list representing the acquaintances of a base object. In an actor system, all entities are actors. Although they may be implemented as local state, even

primitive data objects, such as integers or strings, are considered acquaintances in an actor system. Therefore, in an actor language the *acquaintances* and the *mail queue* comprise the complete state of an actor. The *acquaintances* meta-object allows for check-pointing of actors.

Meta-objects are examined and installed by means of *meta-operations*. Meta-operations are defined in the class called `Object` which is the root of the inheritance hierarchy. All classes in the system inherit from `Object` implying that meta-operations can be called on each actor in the system. The meta-operations `change_mailQueue` and `change_dispatcher` install mail queues and dispatchers for the object on which they are called. Similarly, the meta-operations `get_mailQueue`, `get_dispatcher` and `get_acquaintances` return the meta-objects of a given actor. If no meta-objects have been previously installed, an object representing the built-in, default, implementation is returned. Such default meta-objects are created in a lazy fashion when a meta-operation is actually called.

Transparency and Reuse

By describing our dependability protocols in terms of meta-level dispatchers and mail queues, we are able to construct protocols in terms of operations on messages where we treat each message as an integral entity. There are several advantages to developing dependability protocols in this manner.

The first advantage is the natural way in which protocols may now be expressed. When dependability protocols are described in the literature, they are described in terms of abstract operations on messages, i.e. the contents of the messages are not used in determine the nature of manipulation to be performed. Therefore, it is logical to code protocols in a manner more closely resembling their natural language description.

Secondly, because the protocols are expressed in terms of abstract messages and because every object may have a meta-level mail queue and dispatcher, a library of protocols may be developed which may be used with any object in the system. Such a library would consist of protocols expressed in terms of a mail queue and dispatcher pair. The meta-objects may then be installed on *any* object in the system. Since the protocols deal only with entire messages, the actual data of such messages is irrelevant to the operation of the protocol. Only fields common to every message, such as source, destination, time sent, etc., need be inspected.

The libraries could also be used with other systems, allowing the reuse of dependability protocols. One set of developers could be responsible for the dependability of multiple software systems and develop a protocol library for use with all of them. Since protocols implemented with MAUD are transparent to the application, other development teams, who are responsible for development of the application programs, need not be concerned with dependability. In the final system, protocols from the library may be installed on objects in the application, providing dependability in the composed system.

Example 1: A Replicated Server

In this section, we provide an example of how a protocol may be described using MAUD. In a distributed system, an important service may be replicated to maintain availability despite processor faults. In this section, we will give an example of how MAUD can be used in an actor domain to develop a modular and application-independent implementation of a protocol which uses *replication* to protect against crash failures.

The protocol we describe is quite simple: each message sent to the server is forwarded to a backup copy of the server. In this manner, there is an alternate copy of the server in case of a crash. Reply messages from both the original and backup servers are then tagged and the client eliminates duplicate messages.

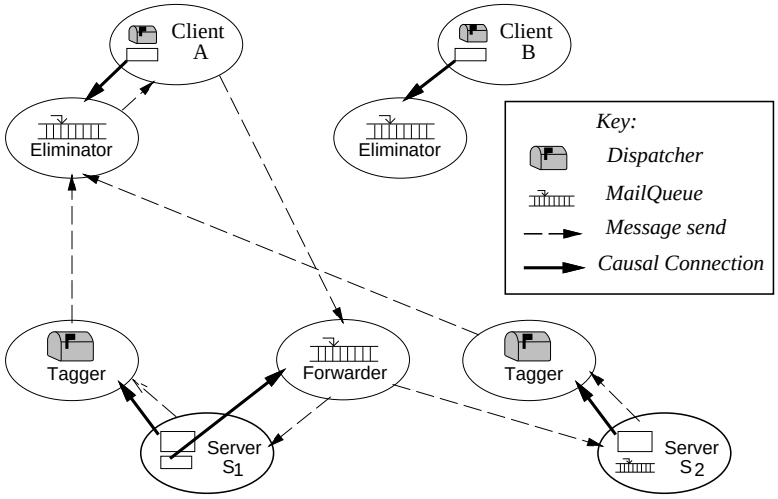

Figure 1.2.2: When a message is sent by the clients A or B to the replicated service S_1, the message is received by the *Forwarder* and a copy is forwarded to the backup S_2. When the servers reply, the *Tagger* dispatchers tag each message so that the Eliminator mail queues may remove duplicate results. If S_1 crashes, manager actors will install S_2 as the new server.

Figure 1.2.2 shows the resulting actions occurring when a message is sent to the replicated service. The original server is actor S_1. When a message is received by the *Forwarder*, the message is forwarded to the backup S_2. S_2 is initialized with the same behavior and state of S_1. Since they will receive the same messages in the same order, their state will remain consistent. Therefore, any replies will be identical and in the same order. The replies are tagged by the dispatchers of class *Tagger* and only the first copy of each message is passed on to the client by *Eliminator*.

Forwarding messages to the backup server is implemented using a meta-level mail queue. The Screed code for this mail queue is presented in Figure 1.2.3. Using a dispatcher, each reply message of the server is tagged to allow the elimination of duplicate replies by the client. A mail queue at the client performs this duplicate elimination. The code for this mail queue is shown in Figure 1.2.4.

```
class Forwarder : MailQueue {
  actor backup;
  actor server;

  put(msg m) {
    m.base_send();
    m.set_dest(backup);
    m.send();
  }
}
```

Figure 1.2.3: Code for the server-end mail queue which implements replication. The mail queue *Forwarder* sends a copy of each message to a backup copy of the server.

```
class Eliminator : Mailq {
  int tag;
  actor members[NUMREP];
  actor client;

  /* No get method is required since we use
   * the default behavior inherited from Mailq */

   put(msg m) {
     int i;

     for (i=0; i < NUMREP; i = i + 1)
       if (m.get_src() == members[i])
         /* Since the message was from a replica,
          * we know that the first argument is a tag and
          * the second is the original message. */
          if (m.arg[0] < tag)
            /* Discard message */
            return;
          else if (m[0] == tag) {
            self.enqueue(m[1]);
            tag = tag + 1;
          }
   }
}
```

Figure 1.2.4: Code for the server-end mail queue which implements replication. The mail queue *Eliminator* removes tags (which have been added to all server replies by some other dispatcher) and takes the first message labeled by a new tag.

We assume that managers themselves install appropriate meta-objects realizing a given dependability protocol. Therefore, we specify the relevant dependability protocols by describing the behavior of the initiating manager as well as the installed mail queues and dispatchers. A manager in charge of replicating a service takes the following actions to achieve the state shown in Figure 1.2.2:

1. The specified server is replicated by a manager by creating an actor with the same behavior and state.
2. A mail queue is installed for the original server to make it act as the *Forwarder* described above.
3. The mail queues of the original clients are modified to act as the *Eliminator* described above.
4. The dispatchers of the servers are changed to tag all messages so that the *Eliminator* may remove copies of the same message.
5. Upon detection of a crash of S_1, the manager takes appropriate action to ensure all further requests to the server are directed to S_2. The manager may also create another backup at this time.

Although this example is simple, it does illustrate some of the benefits of our approach. The manager initiating the replication protocol needs no advance knowledge of the service to be replicated nor does the replicated service need to know that it is being replicated. Because the clients using the replicated service are not modified in any way, this gives us the flexibility to dynamically replicate and unreplicate services while the system is running.

Composition of Dependability Characteristics

In some cases, dependability can only be guaranteed by using several different protocols. For example, a system employing replication to avoid possible processor faults may also need to guarantee consensus on multi-party transactions through the use of three-phase commit or some similar mechanism. Unfortunately, writing one protocol which has the functionality of multiple protocols can lead to very complex code. In addition, the number of possible permutations of protocols grows exponentially — making it necessary to predict all possibly needed combinations in a system. Therefore, it is desirable to be able to compose two protocols written independently. In some cases this may not be possible due to a conflict in the semantics of the two protocols. In other cases, performance may depend greatly on the way in which two protocols are composed. For many common protocols such as replication, checksum error detection, message encryption, or check-pointing, composition is possible.

Because the meta-components of an object are themselves objects in a reflective system, there is a general solution for composing two protocols using MAUD. A simple change to the meta-operations inherited from the `Object` class, along with a few restrictions on the construction of mail queues and dispatchers, allows us to layer protocols in a general way. Figure 1.2.5 shows how an *add-mailq* method could be expressed in terms of the other meta-operations to allow layering.

Because the mail queue and the dispatcher are objects, we can send a message to install meta-objects customizing their mail queue or dispatcher. By adding protocols in the above manner, the outer mail queue functionality will be performed on incoming messages before they are passed on to the "inner" mail queues. For the send behaviors, the process is reversed with the innermost send behavior being performed first and the outermost behavior last, thereby creating an onion-like model with the newest layer closest to the outside world.

```
add_mailq (actor aMailq) {
   if (mailq == nil) {
      self.change_mailq(aMailq);
   else mailq.add_mailq(aMailq);
}
add_dispatcher (actor aDispatcher) {
   if (dispatcher == nil) {
      self.change_dispatcher(aDispatcher);
   else dispatcher.add_dispatcher(aDispatcher);
}
```

Figure 1.2.5: The additional methods which must be inherited to allow for protocol composition.

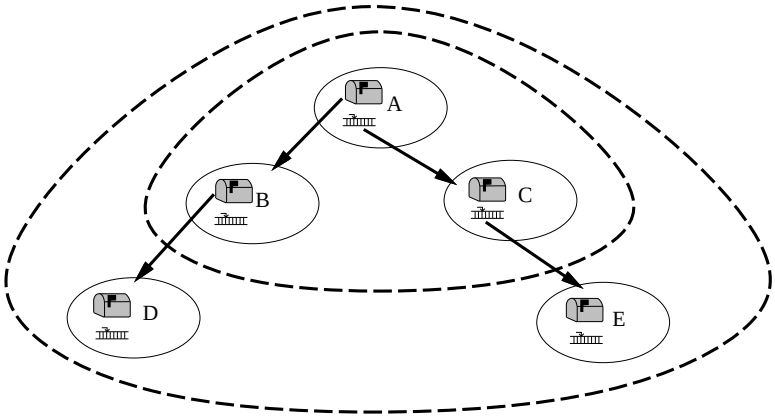

Figure 1.2.6: Partners and Owner relationships. A is the owner of all other actors in the figure. Dispatcher B and mail queue C are partners as well as dispatcher D and mail queue E.

To preserve the model, however, several restrictions must be applied to the behavior of dispatchers and mail queues. We define the *partner* of a mail queue as being the dispatcher which handles the output of a protocol and the partner of a dispatcher as being the mail queue which receives input for the protocol. In Figure 1.2.6, B and C are partners as well as E and D. Each pair implements *one* protocol. It is possible for a meta-object to have a null partner.

The *owner application* of a meta-object is inductively defined as either its base object, if its base object is not a meta-object, or the owner application of its base object. For example, in figure 1.2.6, A is the owner application of meta-objects B, C, D, and E. With the above definition we can restrict the communication behavior of the actors so that:

- A mail queue or dispatcher may send or receive messages from its partner or an object created by itself or its partner.
- A dispatcher may send messages to the outside world, i.e. to an object which is not a mail queue or dispatcher of the owner application (although the message might be sent through the dispatcher's dispatcher). A dispatcher may receive `transmit` messages from its base object and otherwise may only receive messages from its mail queue partner. Therefore, a dispatcher

with a null mail queue partner may *only* receive `transmit` messages from its base object or communicate with actors it created.

- A mail queue may receive messages from the outside world (through its own mail queue) and send `put` messages when responding to `get` messages from its base object. Mail queues may otherwise only send messages to its dispatcher partner or actors it created. Therefore, a mailq queue with a null dispatcher partner may *only* send `put` messages to its base object or communicate with actors it created.
- Objects created by a mail queue or dispatcher may communicate with each other, their creator, or their creator's partner.

Because of the above restrictions, regardless of the number of protocols added to an object there is exactly one path which incoming messages follow — starting with the outermost mail queue — and exactly one path for outgoing messages in each object — ending with the outermost dispatcher. Therefore, when a new dispatcher is added to an object, all outgoing messages from the object must pass through the new dispatcher. When a new mail queue is installed, it will handle all incoming messages before passing them down to the next layer.

Thus, a model of objects resembling the layers of an onion is created; each addition of a protocol adds a new layer in the same way regardless of how many layers currently exist. With the above rules, protocols can be composed without any previous knowledge that the composition was going to occur and protocols can now be added and removed as needed without regard not just to the actor itself, but also without regard to existing protocols. In Figure 1.2.6, actors B and C are initially installed as one "layer." Messages come into the layer only through C and leave through B. Therefore, D and E may be installed with the `add-mailq` and `add-dispatcher` messages as if they were being added to a single actor. Now messages coming into the composite object through E are then received by C. Messages sent are first processed by B and then by D.

Example 2: Composing Two Protocols

Figure 1.2.7 shows the result of imposing the protocol described in Example 1 on a set of actors already using a checksum routine to guarantee message correctness. Originally, each actor had a corresponding `Check-In` mail queue and a `Check-Out` dispatcher. When server S_1 is replicated, its meta-level objects are also replicated. The `Forwarder` mail queue is installed as the meta-level mail queue of S_1's mail queue. It will forward all messages to S_2. A `Tagger` dispatcher is installed for each of the two servers and the `Eliminator` mail queue removes duplicate messages at the client. Although this protocol would be difficult to write as one entity, composition allows their modular, and therefore simpler, development.

In terms of our onion-layer model, each *Check-In*/*Check-Out* pair forms a layer. For example, the innermost layer for server S_1 consists of a *Check-Out* dispatcher and a *Check-In* mail queue. The outermost layer at S_1 is comprised of a *Tagger* dispatcher and a *Forwarder* mail queue. The client A also has two layers. However, its outer layer consists solely of the *Eliminator*: this mail queue has a null dispatcher partner. Similarly, at server S_2, the outermost layer consists only of a *Tagger* dispatcher with a null mail queue partner.

As can be seen in the above example, the onion-layer model only provides consistency for mail queue and dispatcher installation at a single node: a manager that follows the above rules may still install protocols incorrectly. Such an error may occur if the protocols are installed in one order at one node and in a different order at another node. For example, if the manager installed the *Eliminator* mail queue at client A as the innermost layer rather than the outermost, the system would not operate correctly. An area of current research is developing methods for specifying managers which simplify protocol installation and guarantee global consistency.

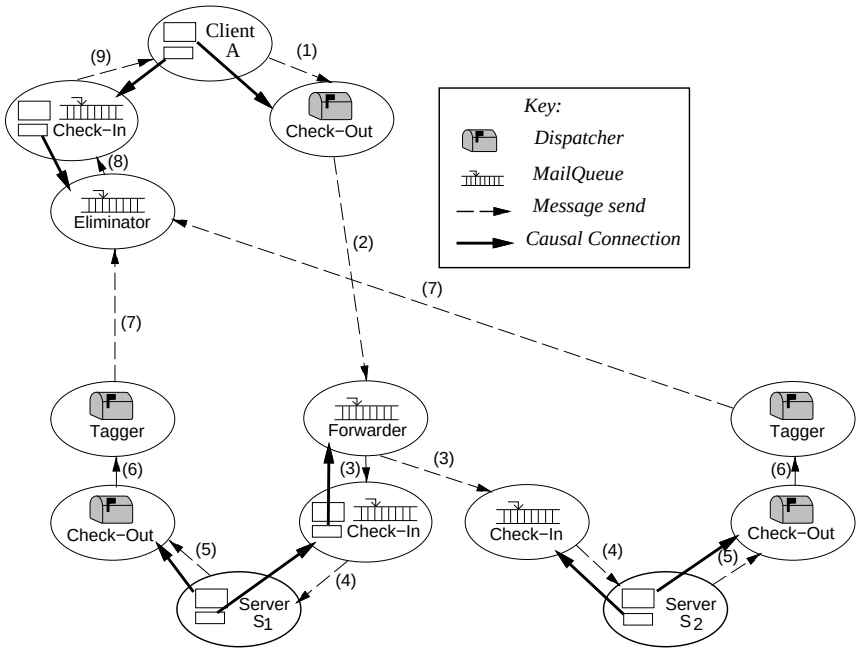

Figure 1.2.7: System resulting from the composition of a replication protocol and message checksum protocol. When a message is sent by the client A (1), the *Check-Out* dispatcher adds the checksum information to the message. The message is then forwarded to the replica as describe in Example 1 (2–3). The checksum information is removed by the *Check-In* mail queue(4) and the messages are processed, resulting in a reply (5). The reply messages both have the checksum (6) information added before they are tagged and sent to the client (7). At the client, duplicate messages are removed, the checksum information is checked, and the message is delivered.

Performance

In implementing MAUD in Broadway, we have found that, in most cases, the additional message overhead caused by reflection is small compared to the actual cost in messages accrued by the protocols themselves. Using MAUD, there is an additional $3n$ messages upon message reception, where n is the number of protocols composed together to provide dependability for a single object. Upon reception of a message, the message is routed up the chain of meta-level mail queues (n messages) and then worked its way down through a series of `get` and `put` messages. For message transmission, there are n additional `transmit` messages.

Since each object is usually protected by only a small (1 or 2) number of protocols, this cost is not great. Since meta-level objects are most likely to be local to the base actor, most messages to meta-objects will be local and inexpensive. Furthermore, we use *caching* of the highest level mail queue to eliminate n of the messages: the system records the address of the top level mail queue and directs all messages intended for the base object to this mail queue. To preserve correctness with caching, meta-object installation is made *atomic*. Once a protocol installation begins, no messages are processed until the protocol is installed.

This optimization is especially critical if some meta-objects need to be on a separate node. Placement of meta-objects on a different node from the base object is only done when physical separation is necessary for dependability: in this case, the inter-node communication from meta-mail queue to base-object or base-object to meta-dispatcher would normally be required by the protocol, regardless of implementation technique. On the other hand, the communication cost from base-object to meta-mail queue is only due to the nature of using reflection. Therefore, caching eliminates this additional expense.

1.2.6 Exception Handling

Given a meta-level such as MAUD, it is still necessary for a programming language to provide flexible constructs supporting adaptive dependability. In particular, it is important to convey information to the correct entities when system failures occur. We have chosen exception handling as the medium through which *managers* are informed of problems in the system. This technique has been used extensively with forward error recovery: we simply extend the notion by having our managers prevent future failures through dynamic protocol installation.

In this section, we describe the exception handling mechanism in Screed, our prototype actor language. To support adaptive dependability, faults and exceptions have been unified as one concept and exception handlers may be shared between objects. Broadway provides a set of system exceptions, some of which are notifications of failures. For example, when an actor attempts to communicate with an unreachable node, a `crash` exception is generated.

We begin with a discussion of the general structure of exception handling in Screed followed by a specific illustration of the syntax used. We then show how this structure may be used with the meta-architecture to design adaptively dependable systems.

Exception Handling Components

Exceptions are *signaled* whenever an unexpected condition is encountered. An exception may be signaled either by the run-time system or by the application. The former are referred to as *system exceptions* and the latter as *user-defined exceptions*.

Exceptions in Screed are represented as objects, as proposed in [13] for sequential object-oriented languages. Although none of the other concurrent languages discussed above have taken this approach, we feel representing exceptions as objects allows for more flexible and efficient exception

handling: all the information needed by a handler is contained in one object. All system exceptions are derived, through inheritance, from the class `exception`. User-defined exceptions may inherit from the exception class or from any other node on the system exception inheritance tree. Below, we discuss the parties involved in the generation of an exception and then the structure of system exceptions.

There are four roles involved in handling any exceptional condition: invoker, signaler, exception, and handler (see Figure 1.2.8). Each role is represented as an object in the system. The *invoker* initiates the method of the *signaler* which results in an exception. The occurrence of an exception generates a signal. When a signal occurs, a new *exception* object is created. The signaler notifies the appropriate *handler* object of the exception's mail address. The handler must then diagnose the exception and perform any appropriate actions for handling the exception.

Exception handlers are constructed by the programmer as Screed actor-classes. For each exception a handler accepts, a method must exist with the same name as the exception and which takes an instance of the exception class as a parameter. In all other ways, handlers are identical to other actor classes: they may have a set of instance variables, inherit from other classes, and may communicate with any of their acquaintances. They may also have other, non-exception methods.

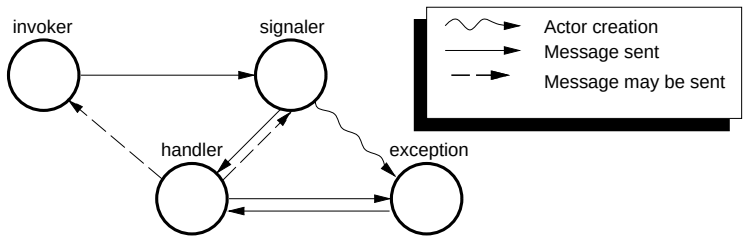

Figure 1.2.8: The four roles involved with an exceptional condition. The *invoker* initiated the method in the *signaler* which caused the *exception*. An object of the appropriate exception class is created and the *handler* is notified of the exception's mail address. The handler may then interact with the invoker and/or the signaler as well as the exception object to resolve the exception.

All exceptions must inherit from the class `exception`. When an exception is signaled, an object of the appropriate exception class is instantiated and initialized with any information needed by the handler to process the exception. Some of the initialization fields are supplied by the run-time system. These fields are contained in the `exception` class from which all exception objects inherit, and are utilized through the methods inherited from the `exception` class.

Additional arguments for the initialization of an exception may be specified by the objects raising a signal. For example, an `arithmetic` exception which is initiated by an application could be initialized when signaled with the values of the operands in the arithmetic operation. This exception object would still have default values specified by the system.

Methods defined in the exception class make use of the system-supplied values. These methods are:

`name` returns the name of the exception as a method value. Since method names are first-class values in Screed, this method enables the automatic calling of the correct method to handle it.

`invoker` returns the mail address of the actor which invoked the method resulting in the generation of the signal.

`signaler` returns the mail address of the signal generator.

`source` returns the name of the method in which the signal was generated.

`arguments` returns a list of the arguments that were passed to the method in which the signal was generated.

`request` returns `TRUE` if the invoker is waiting on a reply, `FALSE` otherwise.

`reply` allows a handler to reply to a request that was interrupted by the signal. The reply method can be used to supply an acceptable value to the invoker, thereby allowing the continuation of the computation.

Each exception handler may utilize only a few of these fields. However, since our environment is asynchronous, we want to preserve all available information. There are no guarantees that this information will be retained by either the invoker or the signaler. Use of exception objects provides us with the flexibility to include a large amount of information without creating complicated function calls or messages: all the information is packed into an object and is referenced through a standard interface. In a procedural approach, long parameters lists would be necessary to achieve the same effect.

Broadway currently supports three different system exceptions. All three inherit directly from the class `exception`. A `bad-method` exception is instantiated when an actor receives a message it cannot processes. The `bad-method` exception class provides the behavior of the destination actor. In general, there is very little the run-time system can do to correct such an error, but this information allows a handler to provide meaningful error messages to the user.

An `arithmetic` exception is generated whenever Broadway traps an arithmetic error. Currently, this exception provides the state under which the exception occurred. We plan to expand this exception to include a string representing the expression being evaluated.

Broadway also provides some failure detection capabilities. Each node on Broadway has a failure detector which uses a *watch-dog timer* approach to detect the failure of, or inability to communicate with, other nodes. A `crash` exception is generated whenever an actor attempts to communicate with an actor on a non-existent or unreachable node. A `crash` exception consists of the original message and the identity of the node which cannot be reached. Notice that, although Broadway has detected a component failure, it is treated similar to any other system exception. It is also possible for an object to *subscribe* to a failure detector. In this case, the subscriber's handler will automatically receive an exception whenever a failure is detected, even if the object did not try to communicate with the failed node.

Besides detecting node crashes, Broadway will also handle the failure of individual actors. If an actor crashes due to an error that is trapped by Broadway, that actor address will be marked as a crash. Currently, only arithmetic errors are trapped by Broadway and, therefore, this is the only manner in which a single actor may crash. If the defunct actor receives a message, a `dead-actor` exception will be generated. The `dead-actor` exception inherits from the `crash` exception. It also contains a reference to the exception generated when the actor crashed. (Currently, this is always an `arithmetic` exception.)

Exception Handling in Screed

In this section, we describe our two syntactic additions to Screed which enable exception handling: the `handle` statement which associates exceptions with handlers, and the `signal` statement which generates an exception.

```
handle (exception1, exception2 with  handler1,
        exception3 with handler2,
        ⋮
       )
{
   /* Any block of code goes here */
}
```

Figure 1.2.9: The structure of a handle block in Screed. *exception*1, *exception*2 are actor class names. *handler* is the name of an object.

In Screed, handlers can be associated with exceptions for either entire actor classes or for arbitrary code segments within a method. Figure 1.2.9 gives the syntax for a `handle` statement. The statement defines a scope over which specific exceptions are associated with a particular handler. If any method invocation contained within the code block of the `handle` statement results in an exception, the signal is routed to the correct handler as specified by the `with` bindings. As explained above, the exceptions are specified as class names and the handlers are addresses of objects.

Handler statements may be nested. In this case, when an exception is generated, the innermost scope is searched first for an appropriate handler. If a handler for the exception does not exist then higher level scopes are checked.

```
handle (arithmetic with arithhandler,
        bad-method with aborthandler) {
   actor A;
   actor B;
   actor E;
   A = new complex(2,3);
   B = A.divide(C);
   handle (arithmetic with myhandler)
      E = B.divide(D);
      myNum = res;
   }
}
```

Figure 1.2.10: An example of handler scopes and their effect. The outermost `handle` statement provides handlers for `arithmetic` and `bad-method` exceptions. The inner statement overrides the outer scope in that all arithmetic exceptions will be handled by `myhandler`.

Figure 1.2.10 demonstrates the scoping rules. In the scope of the outer handle statement, if in computing B (by dividing A by C), an arithmetic exception is generated (possibly by dividing by zero), the signal will be passed to `arithhandler`. The computation of E through the division of B by D, however, is in the scope of the second *handle* statement. Therefore, any arithmetic signals generated by this action are sent to `myhandler`. Conversely, if our complex objects do not have a divide method, our actions will generate a `bad-method` signal which will be handled by `aborthandler`.

Unlike the complaint address based schemes[21, 26], our syntactic mechanisms do not require explicit specification of a handler's address with each message. For any given scope, including a single message send, handlers — our equivalent of complaint addresses — may be specified for each indi-

vidual exception or for any group of exceptions. One handler need not be specified for all exceptions. Additionally, our method takes greater advantage of the available inheritance mechanisms as well as the general structure of object-oriented languages: both exceptions and handlers are expressed as objects in our system.

The above constructs work well within methods. However, there are two levels of scoping above the method level in Screed: the global and class levels. Exception handling at the class level is specified through the use of a `handler` statement which encloses several method definitions. In this manner, exception handling may be specified for an entire class by enclosing all methods in one handler statement. Such a construction does not prohibit `handler` statements inside the methods.

A `handle` statement may *not* be defined across class boundaries as that would require the use of shared variables between class instances. However, to provide exception handling at the global level, Screed supports the system-defined handler class `Default-Handler`. An instance of this class handles all signals which are not caught by another handler. Default system behavior is for a signal to be simply reported to the terminal. `Default-Handler` may be overwritten by a programmer defining a custom class of the same name. In this way, a final level of exception handling may be defined by the programmer. This type of facility is especially useful for writing debuggers. Any exception not defined in a custom `Default-Handler` class is handled by the system-default. Note that the system creates only one instance of the `Default-Handler` class: all otherwise unhandled signals are delivered to this instance.

As mentioned previously, exceptions may be generated as user-defined signals. A signal is generated by a `signal` statement.

`signal` *exception-class-name*(*args*...);

The `signal` statement generates a message to the appropriate exception handler. The arguments are used for initialization of the exception as defined by the interface of the particular exception class. The `signal` does not interrupt the flow of control in the code, although a Screed `return` statement could follow the `signal` to end the method.

In many cases, it is necessary for the signaler of the exception to await a response from the handler before proceeding. `signal` statements are treated, syntactically, as message sends to a handler. Therefore, `signal` statements may act as an remote procedure call in the same manner as Screed message-sends. Thus, the handler may return a value to be used by the signaler. Such a case would be:

```
res = signal div-zero();
```

For this example, the exception handler would return an actor address as the value *res*. Then, the rest of the signalling method may compute.

In other systems, a special construct exists for generating signals within the current context, i.e. generate a signal which is caught by the `handle` statement in whose scope the statement occurs. An example of such a construct would be the `exit` statement in Clu [24]. In Screed, such a construct in not necessary: the actor can explicitly send a message to the appropriate exception handler.

1.2.7 Supporting Adaptive Dependability

A significant difference between exception handling in Screed and other languages is the use of third-party exception handlers. In languages such as CLU [24], SR [20], and Ada [12], exception handling routines are defined within the scope of the invoking objects. We refer to this approach as two-party exception handling (the invoker and the signaler) and our approach as three-party

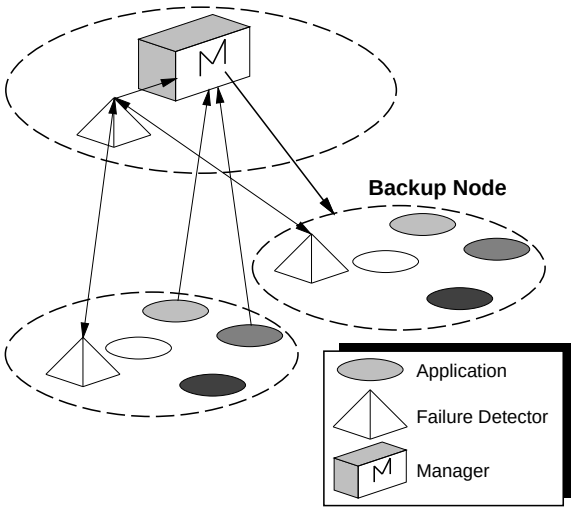

Figure 1.2.11: A prototypical adaptively dependable system. In this case, the manager M receives input in the form of exceptions from application objects and notices from the failure detector. Upon determination that the *Primary Node* is unstable, M allocates the *Backup Node* and creates the appropriate objects to replace the *Primary Node*. Note that, in actuality, M is probably a replicated entity to ensure its resilience and availability.

exception handling (the invoker, the signaler and an independent handler). We have found that two-party exception handling is unsatisfactory for modeling the complex relationships between objects such as those required for adaptively dependable systems.

The key difference between two- and three-party systems is in the potential *knowledge* of a given object. With two-party exception handling, the exception handler, which is also the invoker, may know only of exceptions its invocations generated. Therefore, in such a system it is very difficult to develop a global picture of the fault pattern in the system. In a three-party system, such monitoring may be naturally expressed in terms of the exception handler since it may handle exceptions for a large group of objects or even the entire system. Furthermore, an autonomous exception handler may subscribe to any available failure detectors, thereby augmenting the knowledge received through exception signals.

A third-party exception handler may also be designated as an object with special rights. In this manner the system may be safely modified in response to exceptions and failures. Since it is dangerous to allow the arbitrary modification of one actor by another, most two-party systems can express reconfiguration of the system only by mimicking a three-party system, i.e. they must notify a third object with special rights of the exceptions they encounter and this object may then reconfigure the system.

Thus in adaptively dependable systems, the resulting system architectures will look quite similar to Figure 1.2.11. Such a system may allow the dynamic installation of dependability protocols or may simply support the reconfiguration of several objects in response to exceptions. In either case, the system will have a *manager* with special rights to modify other objects. The manager will act as the exception handler for some group of objects being monitored. Furthermore, the manager may also subscribe to failure-detection services. Upon receiving enough input to determine that some unacceptable condition exists, the manager reconfigures the group over which it has authority.

Example 3: Reconfiguration to Preserve Failure Semantics

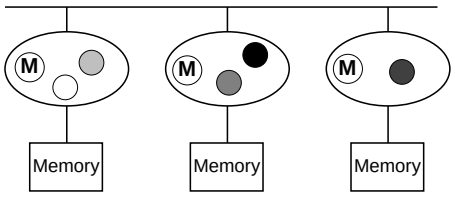

Figure 1.2.12: This figure shows the topology of the network on a satellite. The shaded circles represent actors and the manager for memory errors is denoted by the replicated actor **M**. The memory uses checksums to avoid returning incorrect values. Instead, if an incorrect value is found, a special exception value is returned which may then trigger an exception in the application. The manager is replicated to guarantee a high level of fault-tolerance.

To illustrate the concepts we described above, consider a distributed system which is operating in a hostile environment. A good example of such a system is a satellite with multiple processors, each with its own memory. Assume that the memory of these processors was developed to never return incorrect data to a *read*. Instead, the memory will detect the error through some checksum algorithm and return an error condition value. This reserved value can then be used to signal an exception and initiate forward error recovery. The specifications for this system state that such memory errors should occur with probability 10^{-8}. Such a system is shown in Figure 1.2.12. Notice that the *manager* is itself replicated to ensure fault-tolerance in this vital system component.

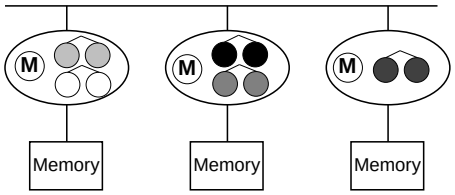

Figure 1.2.13: The applications on each node have now been replicated using the meta-level protocol described in Example 1. Since we are protecting only against the corruption of individual memory locations, the replica may reside on the same node as the original actor, thereby improving the performance of the protocol.

Once the system is launched, these memory components seem to operate correctly and the *manager* responsible for memory errors occasionally performs forward error recovery on the objects. However, the rate at which these memory errors are occurring is unacceptably high (10^{-5} faults/read) and system performance degrades significantly due to repeated memory faults and subsequent error recovery. Therefore, the manager installs the replication protocol described in Example 1. The resulting system is shown in Figure 1.2.13. Since both the original actor and the replica will be reading values from different memory locations, the probability that a memory error will be noticed is now 10^{-10}, well within the specified tolerance. When an exception occurs, the manager will still have to perform some corrections, but the system can keep computing during this time and the error recovery will be simplified due to the existence of the replica. Considering the nature of the faults, the replica may be placed on the same node as the original actor. However, if instead of signalling

an exception, nodes crashed when they could not read memory, the replicas would have been placed on different nodes.

1.2.8 Conclusion

In this paper, we have described a methodology for the development of adaptively dependable systems. Adaptively dependable systems may function over a long duration despite a changing execution environment. Whether the changes are due to a variance in the components comprising the system or to a change in the physical environment in which the component operates, the use of dynamic protocol installation combined with exception handling allows fault tolerance to be guaranteed by the system.

Dynamic protocol installation is enabled through the use of a meta-level architecture. Our meta-level, MAUD, allows the customization of an object's communication behavior on a per-object basis. By describing protocols in terms of modifications to the communication behavior, protocols may be dynamically installed on objects as necessary. Furthermore, if a protocol is no longer required, it may be removed. Through the use of caching and atomic protocol installation, meta-level description of protocols may be implemented with a minimal cost in performance.

We also support composition of protocols. Provided there are no inherent semantic conflicts between two protocols and both protocols are implemented using MAUD, these two protocols may then be composed without foreknowledge that a composition may occur. In this manner, protocols may be constructed in a modular fashion and later combined to provide the desired level of fault-tolerance.

To provide adaptive dependability, we combine dynamic protocol installation with exception handling. We make extensive use of *third-party* exception handlers which are shared between multiple objects. Since these handlers have privileges to modify meta-level objects, they are termed *managers*. A single manager will be informed of all exceptions related to a particular problem. The knowledge may be augmented through subscription to failure-detection services. In this manner, the manager will have all information necessary for a correct diagnosis of fault patterns.

The concepts described in this chapter have been implemented on Broadway — our actor run-time system — and are accessed through the language Screed. Screed provides exception handling constructs which support managers and provides access to the meta-level architecture implemented in Broadway.

One problem not solved by our framework is guaranteeing the consistent installation of protocols on multiple actors. We are currently developing a *Protocol Description Language* which will allow a protocol to be expressed as a single entity. A protocol compiler can convert these protocols into MAUD mail queues and dispatchers. To address the installation problem, the compiler creates objects that guarantee correct installation of the protocol. Managers use these objects to install protocols on all actors involved in the protocol.

Acknowledgments

The research described in this paper has benefitted from earlier collaboration with Svend Frølund and Rajendra Panwar. The authors would also like to acknowledge helpful comments from Christian Callsen, Svend Frølund, WooYoung Kim, Rajendra Panwar, Anna Patterson, Shangping Ren, Carolyn Talcott, Nalini Venkatasubramaniam, and Takuo Watanabe, among others.

References

[1] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Developement. In *USENIX 1986 Summer Conference Proceedings*, June 1986.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[3] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[4] 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.

[5] G. Agha, I. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computation. In R. Cleaveland, editor, *The Third International Conference on Concurrency Theory (CONCUR '92).* Springer–Verlag, 1992. LNCS (forthcoming).

[6] Gul Agha and Christian Callsen. ActorSpace: An Open Distributed Programming Paradigm. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993. Also published as a Special Issue of SIGPLAN Notices vol. 28, No. 7, pages 23–32, July 1993.

[7] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[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] E. Cooper. Programming Language Support for Multicast Communication in Distributed Systems. In *Tenth International Conference on Distributed Computer Systems*, 1990.

[10] Antonio Corradi, Paola Mello, and Antonio Natali. Error Recovery Mechanisms for Remote Procedure Call-Based Systems. In *8th Annual International Phoenix Conference on Computers and Communicaton Conference Proceedings*, pages 502–507, Phoenix, Arizona, March 1989. IEEE Computer Society Press.

[11] Flaviu Cristian. Understanding Fault-tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, 1991.

[12] Quian Cui and John Gannon. Data-Oriented Exception Handling in Ada. *IEEE Transactions on Software Engineering*, 18:98–106, May 1992.

[13] Christophe Dony. Improving Exception Handling with Object-Oriented Programming. In *Proceedings of the 14th Annual International Computer Software and Applications Conference*, pages 36–42, Chicago, 1990. IEEE Computer Society, IEEE.

[14] Christophe Dony, Jan Purchase, and Russel Winder. Exception Handling in Object-Oriented Systems. *OOPS Messanger*, 3(2):17–29, April 1992.

[15] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *CAMELOT AND AVALON: A Distributed Transaction Facility.* Morgan Kaufmann Publishers, Inc., 1991.

[16] Jacques Ferber and Jean-Pierre Briot. Design of a Concurrent Language for Distributed Artificial Intelligence. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 755–762. Institute for New Generation Computer Technology, 1988.

[17] S. Frølund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In *Proceedings of ECOOP 1992*. Springer Verlag, 1992. LNCS 615.

[18] S. Frølund and G. Agha. A Language Framework for Multi-Object Coordination. In *Proceedings of ECOOP 1993*. Springer Verlag, 1993. LNCS 707.

[19] John B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[20] Daniel T. Huang and Ronald A. Olsson. An Exception Handling Mechanism for SR. *Computer Languages*, 15(3):163–176, 1990.

[21] Yuuji Ichisugi and Akinori Yonezawa. Exception Handling and Real Time Features in an Object-Oriented Concurrent Language. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, pages 92–109. Springer-Verlag, Oxford, UK, September 1989. LNCS 491.

[22] Wooyoung Kim and Gul Agha. Compilation of a Highly Parallel Actor-Based Language. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1992.

[23] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[24] Barbara Liskov and Alan Snyder. Exception Handling in Clu. *IEEE Transactions on Software Engineering*, 5(6):546–558, November 1979.

[25] P. Maes. Computational Reflection. Technical Report 87-2, Artificial Intelligence Laboratory, Vrije University, 1987.

[26] Carl Manning. ACORE: The Design of a Core Actor Language and its Compiler. Master's thesis, MIT, Artificial Intelligence Laboratory, August 1987.

[27] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A communication Substrate for Fault-Tolerant Distributed Programs. Technical Report TR91-32, University of Arizona, Tucson, 1991.

[28] M. H. Olsen, E. Oskiewicz, and J. P. Warne. A Model for Interface Groups. In *Tenth Symposium on Reliable Distributed Systems*, Pisa, Italy, 1991.

[29] 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.

[30] Richard D. Schlichting and Titus D. M. Purdin. Failure Handling in Distributed Programming Languages. In *Proceedings: Fifth Symposium on Reliability in Distributed Software and Database Systems*, pages 59–66, Los Angeles, CA, January 1986. IEEE Computer Society Press.

[31] Santosh Shrivastava, Graeme Dixon, and Graham Parrington. An Overview of the Arjuna Distributed Programming System. *IEEE Software*, pages 66–73, January 1991.

[32] B. C. Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, Massachusetts Institute of Technology. Laboratory for Computer Science, 1982.

[33] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[34] 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.

[35] C. Tomlinson and V. Singh. Inheritance and Synchronization with Enabled-Sets. In *OOPSLA Proceedings*, 1989.

[36] 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.

[37] T. Watanabe and A. Yonezawa. A Actor-Based Metalevel Architecture for Group-Wide Reflection. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 405–425. Springer-Verlag, 1990. LNCS 489.

[38] C. T. Wilkes and R. J. LeBlanc. Distributed Locking: A Mechanism for Constructing Highly Available Objects. In *Seventh Symposium on Reliable Distributed Systems*, Ohio State University, Columbus, Ohio, 1988.

[39] 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.

[40] 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.