Dynamic Module Replacement in a Distributed Programming System

by

Toby Bloom
S.M., Massachusetts Institute of Technology (1979)
B.S., State University of New York at Stony Brook (1974)

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

June 1983

Signature of Author . . . . . .

Department of Electrical Engineering and Computer Science

March 27, 1983

Certified by . . . . . .

Thesis Supervisor

Accepted by . . .

Chair, Departmental Committee

Dynamic Module Replacement in a Distributed Programming System

by

Toby Bloom

Submitted to the
Department of Electrical Engineering and Computer Science
on March 28, 1983 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

The replacement of parts of software systems is an important aspect of programming methodology. Most of the research in this area has centered around support for modular construction and the clear separation of interface from implementation. The emphasis has been on producing easily modified static program structures.

With the recent increased interest in distributed systems, attention has been focussed on a class of applications for which this approach to modifiability is insufficient. These are applications involving long-running, distributed computations with long-term, on-line state information.

In the context of the Argus programming system, we examine a method of supporting dynamic modification of software for this class of applications. We determine the appropriate granularity of replacement in relation to the module structure of the language, examine the constraints imposed on dynamic replacement by the need to ensure behavioral consistency across replacements, and then analyze functional requirements for a replacement mechanism.

Thesis Supervisor: Barbara H. Liskov
Title: Professor of Computer Science

Acknowledgments

I would like to thank my advisor, Barbara Liskov, for her technical guidance, editorial advice, and above all, her patience over the last eight years (and for reading the final draft in one day). David Reed provided invaluable assistance on the systems aspects of the thesis. I cannot thank John Guttag and Bill Weihl enough for the time and effort they devoted to helping me develop the formal model.

Numerous people contributed to the effort to complete this thesis. In particular, I'd like to thank Tim Anderson for proofreading the draft, and Tamiko Thiel for lending me her terminal for a much longer time than she expected.

The women's community at MIT has been an indispensible source of advice, support, and companionship. My special thanks to Candy Sidner, Jeanne Richard, and Sandy Yulke for their efforts in building that community and welcoming me into it. The many hours of discussion with other women here, especially Deborah Estrin, Julie Lancaster, Liza Martin, Karen Sollins, and Kären Wieckert provided valuable political insights, as well as wonderful diversions.

There are many other people who deserve thanks for helping to keep me sane over the last several years. Stan Zdonik is entirely responsible for my current addiction to bluegrass; any delay that diversion may have caused in the completion of this thesis was of course worthwhile. Steve Berlin has done his best to convince me to keep my priorities straight; my apologies to him for making the task impossible. I could not have maintained the frenetic pace of the last several months without Dan Brotsky's friendship and support; I truly appreciate his being around to listen to me gripe. I would especially like to thank Deborah Estrin for all of her advice, concern, and encouragement.

Finally, I'd like to thank my father for his continued moral, as well as financial, support and for enduring with such good humor his daughter's perpetual student status.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75--C-0661, and in part by IBM under the IBM Graduate Fellowship Program.

Table of Contents

Table of Figures

Chapter One

Introduction

1.1 Motivation

In any software system that exists for a long period of time, it periodically becomes necessary to make changes to various components of the system. While it has long been understood that support for software maintenance is an important part of programming methodology, the proper tools for performing that maintenance easily and correctly are often not available. This thesis examines an aspect of software modifiability not often addressed: the need to incorporate software changes in long-running programs without losing the current state of those programs, and with minimal disruption to the programs' users.

There are a number of reasons that might necessitate replacing a currently running module with a new version. One possible reason is to correct bugs that have become evident. Another is to improve efficiency. (Often optimizations are made based on patterns of use, and these patterns change over time, requiring changes to improve efficiency.) Another occurrence that would necessitate replacement of a module is a modification of the interface to a module being used. Finally, users' requirements change over time, and the implementor of a module may decide to accommodate those changing needs by modifying the module. All of these cases are examples of maintenance problems faced regularly with any large software system that has existed for a long period of time.

Support for software modifiability can be static or dynamic. Static support involves aid to the programmer in constructing software that is easy to modify. Much of the

work in programming methodology, and language design for reliable software has centered on this aspect of support. [11, 17, 13]. Research efforts to study the applicability of those methodologies to distributed programming are currently underway [12]. While we depend on many of the results of these efforts in the work presented here, we are addressing a different aspect of the problem.

Dynamic modifiability is the ability to incorporate software changes in running instances of a program. The need for dynamic modification arises primarily from the existence of long-running programs that maintain long-term, on-line state. If a program were assumed to run for some relatively short period of time, dynamic modification would be unnecessary. Modifications could be installed by waiting until the current "run" ends, and switching to the updated version of the program before the next run. Similarly, if a continuously running program has no long-term state, changes can be installed by allowing the current module to continue executing, but starting the new module in parallel and directing new requests to the new module.

When the execution of a program is continuous, rather than separated into distinct, relatively short "runs", and the module has state that must be retained across requests (as for example in a database system that is always on-line, and handling concurrent and overlapping requests), the modification techniques described above no longer suffice. There is no time at which implementations can be switched, and the new one cannot be started in parallel with the old because of consistency constraints on the state: the state reflects the history of the module, and if two instances are running in parallel, neither's state will reflect all of the requests that have been serviced. In a system that provides robustness and consistency guarantees on the state values in modules, it is essential that the state be preserved accurately across updates to the software. Any technique used must install the changes instantaneously with respect to uses of the module.

In this dissertation, we will examine how modifications can be incorporated in continuously running modules that have long term, on-line state with consistency constraints. The context for this work is the Argus programming system [12]. The choice of the Argus system is important for a number of reasons. First, the Argus language provides the support for static modification mentioned earlier; it therefore allows us to concentrate on the issues of dynamic modification of software that is considered "easily modifiable" by current standards. Secondly, Argus is designed to support the construction of distributed software, thus providing the context for examining dynamic replacement of modules that span nodes of a distributed system. Finally, Argus is transaction-oriented. The strong consistency and robustness guarantees on program state provide a clear definition of the state that must be preserved across replacements.

1.2 Goals

We have several goals in investigating dynamic replacement. The first is to examine the ramifications of dynamic replacement of continuously running modules in the presence of atomic transactions and consistency constraints on state. The questions we address here include: when can replacement be performed; under what conditions is the replacement of one running module with another considered to be correct; and what constraints should be placed on dynamic replacement to ensure that the assertions upon which clients of a module depend are not undermined by the replacement? Our goal is not to provide automatic verification of replacements, but to define the conditions precisely enough that implementors performing replacements will understand the issues involved and be able to convince themselves that a replacement is correct.

Our research has shown that the conditions in which replacement can be correctly

achieved are far more restricted than previously thought. In Chapter 4, we present a model in which we define those conditions; we also illustrate the cases in which replacement cannot be performed safely. We believe the analysis of legality conditions for replacement is one of the primary contributions of this work.

Our other major goal is to determine how dynamic modification can be integrated into a distributed, transaction-oriented programming system such as Argus. One important question we address is whether support for dynamic modification can be incorporated in a system such as Argus without changing the Argus language, or whether specific language support is warranted, as it is for static modifiability. The approach we will take here is to design a mechanism assuming no changes to the language and use that design to evaluate whether any weaknesses could be alleviated with appropriate language support. Our analysis of the mechanism falls into two parts. We first analyze the relationship of the module structure provided by Argus to the modularity requirements of dynamic replacement. We determine the units of replacement to be supported for Argus programs, and the system support needed to recognize those units. The second aspect of of our work on replacement mechanisms is the analysis of the functional requirements for performing replacement, and the design of a mechanism to support those requirements.

There are a number of specific goals that apply to the design of a mechanism to perform dynamic software modification in the context of Argus. These are primarily concerned with supporting the overall methodological goals of the Argus project. One of the principles of the methodology on which Argus is based is a clear separation of abstraction from implementation [11]. Abstractions provide an explicit interface; the means of implementing the functions provided in that interface are hidden from clients using the abstraction, and should be irrelevant to correct use. This principle implies that it should be possible to replace a running module with another instance that uses a different implementation but provides the same

function, and clients should not have to be notified of the change. In general, replacement should be invisible to clients; all effects should be local to the module(s) being replaced.

Another objective is to make replacement as generally applicable as possible. Since modifications are needed regularly, restrictions on the kinds of modules that can be replaced should be minimized. In keeping with our decision not to change the language, we have made the assumption that no specific additions need be made to modules to make them replaceable.

Finally, we must evaluate the impact of dynamic replacement on the reliability of Argus software. If replacement is to be considered a standard part of the Argus programming environment, we must determine whether it can be included without negatively affecting the reliability of Argus programs. As will be seen later, it may be difficult to allow replacement without introducing additional likelihood of error.

1.3 Related Work

This work draws on a number of similar areas of research, primarily dynamic type replacement [5], [7], and dynamic hardware reconfiguration [14]. This section discusses the relationship of our work to these other areas. As mentioned earlier, this work is also strongly grounded in related work in programming methodology [11] and language design for distributed systems [12].

1.3.1 Dynamic Type Replacement

Both Habermann [7] and Fabry [5] address the dynamic type replacement problem. The problem is the following. In any system using data abstraction, objects of user defined types are created and passed to various modules throughout the system.

The type manager (or *cluster* in CLU terminology) knows and depends upon the internal representation of those data objects. If the implementor of that type wants to make a change in the type manager, a serious difficulty arises: the operations of the new type manager will not be able to interpret the representation of the already existing objects. Furthermore, it is at least expensive, if not impossible, to locate all of those objects at the time at which the new type manager is installed. The solution is to place a version number in each object, and when an operation of the type is invoked on that object, the version number is checked. If it does not correspond to the current version of the type, a translation routine is called to convert the object to the new format before the invoked operation is performed. In Fabry's scheme, a translation routine must exist between every outdated version and the latest version, since there is no way to know when all pre-existing objects have been accessed and therefore converted to the new format. In Habermann's scheme, routines can be provided to translate every version to a canonical form and to translate the canonical form to the latest version. If many versions of objects exist concurrently, the latter reduces the number of routines needed.

This work has obvious similarities to the thesis problem presented here. Both deal with the problem of changing implementations of user defined types dynamically, without losing the values of existing objects. However, [5] and [7] deal with different aspects of the problem than the ones on which we are concentrating. The first distinction is that the focus of this dissertation is defining the semantics of replacement and the conditions under which dynamic replacement can be performed safely, whereas the focus in both [5] and [7] is on defining the mechanism.

There are also differences between our work and the earlier work, with respect to the focus of the mechanism design. The earlier work is concerned with locating all objects of the type because the type manager that accesses the representation of

those objects is changing. The Argus modules with which we are concerned are more like Simula classes than these type managers: all of the operations that directly access the module's data are inside the module. Therefore, not all instances of a given type must be replaced at once. In fact, for most of the cases we have examined, the implementor will want to change only one instance to a new implementation. While that instance must be located, the implementor will be doing that, with the assistance of the replacement system, prior to the replacement. Thus, there is no need to check at each invocation or perform translations of objects as an ongoing system process.

On the other hand, the actual translation routines to change the state from one format to another will often be much more complex than in the type replacement case. The construction of those routines is assumed to be trivial in type replacement and is not addressed at all by Fabry or Habermann. We are concerned with modules that are more complex; the translation from one format to another will be nontrivial. Much of the work in this thesis revolves around providing appropriate user support for writing these translation routines and helping the user to convince himself or herself that the translation is correct.

We also have the additional problem of a distributed system. A substantial amount of user support is required to provide a straightforward means of accessing and moving state objects between nodes in the network. We are also working in an environment in which concurrent processes share access to the state objects. Thus, the two branches of research are related. The work described here does not deal with replacing all instances of a given data type across the system. There are cases in which such a replacement is required, and the solutions provided by earlier work can be used in conjunction with our replacement server.

1.3.2 Dynamic Hardware Reconfiguration

The need to replace hardware modules in a running computer without having to bring down the entire system was recognized long ago. This problem was addressed by Schell [14]. The way in which one hardware module can be swapped for another bears much similarity to the way in which running software modules must be replaced, in part because both are active. Schell's work differs from ours in that there is no state in the hardware modules that must be carried over to the replacement and kept consistent. In most cases, a new module can be installed while the old one is still processing requests. Any new requests will be directed to the new module, while requests in progress can continue in the old module. Thus, there is a notion of quiescing the replaced module before removing it. We are unable to make use of this approach because we must maintain the consistency constraints on the long term state in the modules we are replacing. If the state encodes information about the history of requests, it may not be possible to have the new and old version running simultaneously at any time; replacement must be instantaneous relative to use of the module. Schell also spends a great deal of time ensuring that his method works for replacing the hardware modules that are running the reconfiguration. We have not addressed that problem. Neither the replacement server nor certain other parts of the Argus system (in particular the transaction mechanism, see section 2.2) can be replaced using the mechanism we propose. Since neither carries over state between transactions, they could be replaced with an algorithm similar to Schell's.

1.4 Plan of Thesis

The remainder of the thesis is organized as follows. Chapter 2 describes the Argus language and those features of the system with which the reader should be familiar before the discussion of replacement in the context of Argus. Chapter 3 analyzes the Argus module structure and decides on the appropriate units of replacement to

be supported by a replacement system for Argus. Once the unit of replacement has been decided upon, we define (in Chapter 4) the meaning of correct replacement and the scope of substitutions that can be supported for this unit without undermining the reliability of the Argus system. The relationship that must hold between the module being replacement and its replacement is examined in detail. Chapter 5 deals with users' requirements in performing replacements. We categorize the kinds of changes that will be incorporated using dynamic replacement, and the support users will need in effecting those replacements. Chapter 6 presents the basic mechanism that will be needed to support replacement in Argus. Finally, in Chapter 7 we present our conclusions about the power and usability of such a mechanism, discuss the implications of our results for programming languages in general, and suggest several possibilities for improving support for replacement by extending the language.

Chapter Two

Background: The Argus Language and Library

The work described in this dissertation was done in the context of the Argus distributed programming system [12]. Argus is an integrated programming language and system designed to support construction of well-designed distributed software. It is based on the CLU programming language [11] and brings from CLU an emphasis on encouraging a programming methodology that aids in producing reliable, easily maintainable software. The focus of the Argus development project has been support for applications that are long-lived and have long-term on-line data. The system is assumed to run on a geographically distributed network with heterogeneous nodes. This chapter presents a brief overview of the language, and describes those features that must be understood to define replacement of modules in Argus programs. We concentrate on the overall structure of Argus programs, rather than on specific language constructs. We also describe aspects of the Argus implementation that are relevant to the design of the replacement mechanism. Much of the description given here is taken from [12] and a more detailed description can be found there.

2.1 Argus Modules

Programs in Argus are composed of modules called *guardians.* A guardian provides some service or encapsulates some resource. Internally, a guardian contains a set of processes and a set of objects. Though many guardians can exist at the same node, any single guardian exists entirely at one node in the system. Each guardian has an independent address space; only its processes may directly access the guardian's

objects. Guardians communicate with one another solely via message passing. Hence, a guardian can be thought of as a *logical node* in the system. Communication among guardians is location independent: clients of the guardian need not know at which physical node a guardian resides.

Figure 2-1 shows the general structure of a guardian.

Figure 2-1: Structure of a Guardian Definition

name = **guardian** [parameter-decls] **is** *creator-names*
handles *handler-names*

{abbreviations}
{ [stable] state-variable-decls-and-inits}

[**recover** *body* **end**]
[**background** *body* **end**]

{creator-handler-and-local-routine-defn's }

end *name*

Notation: [] signifies term is optional
{} signifies term can appear 0 or more times

Handlers are the remotely invocable operations provided by a guardian. Clients (i.e. other guardians) make use of a guardian by sending an invocation message to one its *handlers.* A handler's type specification consists of its name, along with the types of its arguments and return values.

In addition to the set of handlers, the definition of a guardian type contains a set of operations called *creators.* A creator has the effect of bringing a guardian of the type into existence, in addition to executing the user-defined code in the creator body. The creator usually performs the function of initializing the guardian's *state.*

Thus, a guardian's interface to the rest of the system consists of a set of handlers and

a set of creators. Associated with each guardian type is a *guardian interface type* that is automatically created by the Argus system, and has get_*handler* operations for each handler in the guardian interface. (The interface object does not contain the creators.) Whenever a guardian instance is created, the Argus system creates an interface object containing the id's of the guardian's handlers. It is this object that clients of the guardian gain access to.

The *state* of a guardian is the set of objects in the guardian that can be shared by all the handlers. There may also be objects that are local to a single process. The state is divided into *stable* and *volatile* objects. The stable objects, i.e. those pointed to by the *stable* variables, are guaranteed to survive crashes; consistent copies of this stable state are written to stable storage [10] upon the commit of atomic actions, which will be described in section 2.2. Upon recovery from a crash, a copy of the state is automatically retrieved from stable storage, and all nonstable objects are reinitialized by the guardian's *recovery code.* One aspect of guardian state that is critical to replacement is that the types of state objects used, and the form of stable data, is entirely implementation dependent. Two different implementations of the same guardian type need not have the same types of stable or volatile state objects.

Finally, there is a section of the guardian text known as the *background code.* Background code runs as a process (or processes) after guardian creation or recovery (unlike handlers, which run only in response to clients' invocations). It may run for the life of the guardian, or terminate after performing some specific task. Thus, a guardian can function as an independent entity in addition to responding to requests.

In addition to guardians, Argus recognizes all the CLU module types: clusters, procedures and iterators. However, instances of these types always exist within guardians in Argus. Different guardians may use different implementations of the

same data or procedural abstraction concurrently. Since the guardians communicate only through message passing, and use an external representation mechanism for communicating abstract values [9], the use of different implementations at different guardians creates no problems. In a sense, guardians function as abstract processors, each having an independent address space and (at least semantically) distinct copies of all code.

2.2 Atomicity

As mentioned earlier, Argus guarantees that a consistent copy of a guardian's state will survive crashes. It further guarantees that if an activity of an Argus program updates data at many guardians, and there are consistency constraints among those guardians, then consistency can be maintained across all of those guardians' states, regardless of their locations in the system, and in spite of crashes. All of those states will be updated in stable storage at completion of the activity, or none of them will be.

These guarantees are supported in Argus through the use of *atomic actions* and *atomic objects.* Atomicity consists of two properties: indivisibility and recoverability. Indivisibility means that no other action can see intermediate results of an atomic action. Recoverability means that an atomic action either runs to completion and *commits*, or it *aborts* prior to completion and there are no visible effects from the action. Atomic actions are thus similar to transactions in database systems [6].

Atomic objects, like all abstract objects, may be accessed only through the operations of their type. The operations of an atomic type provide indivisibility and recoverability for the calling actions in the following way. Each operation of the

type is classified as either a read or write operation. Invocation of the operation will cause the appropriate lock to be acquired and held by the calling action until that action commits. All changes to atomic objects are made on a new copy of the object. If the calling action commits, the new copy is retained; if the action aborts, the new copy is discarded and the previous version of the object is retained. A set of basic atomic types, corresponding to the basic types in CLU, is provided by Argus. The example we provide at the end of this chapter makes use of atomic arrays, for example. Argus also supports user-defined atomic data objects.

All atomic actions are serializable, i.e. they appear to have run sequentially. A two-phase commit protocol is used to ensure that the participants in an atomic action uniformly agree to either commit or abort.

Finally, actions may be nested. An atomic action can contain any number of subactions. A subaction can abort without its parent aborting, thus providing a means of isolating failures. When a subaction commits, its parent action can see the results from that subaction, but the changes made by the subaction will not be visible to unrelated actions, and will not become permanent, until the top-level parent action commits. Abort of a parent will undo the effects of a committed subaction. A parent action cannot run concurrently with its subactions, although the subactions may run in parallel.

2.3 Communication

The primary means of inter-guardian communication in Argus is *remote procedure call.* Remote procedure calls invoke handlers of other guardians; they differ from local procedure calls in that remote calls use call-by-value semantics, whereas local calls use call-by-sharing [11]. The invocation occurs as a subaction of the caller's

action. Thus, a handler call will commit or abort, just as other actions do. Upon receipt of a handler invocation, a process is forked in the guardian to execute the handler's code; the process exists only for the duration of the handler execution.

Creators, like handlers, run as subactions of the calling action. The created guardian comes into existence immediately, but does not become permanent until the top-level ancestor action of the creator commits. If that action aborts, the guardian will disappear. Any guardian may also be sent a *terminate* request. Terminate will cause the guardian to be destroyed when the action requesting terminate is committed. However, the immediate effect of *terminate* is to make the guardian appear crashed: invocations from ancestors of the terminating action will abort; invocations received from other actions will be delayed until the terminating action completes (either to top-level, or to an ancestor of the guardian's creator).

One issue that arises in defining remote procedure call is the transmission of abstract objects. Different guardians can use different implementations of a data type, and hence objects of the type in the two guardians will have different representations. It is therefore not possible to send an object directly from one guardian to another. In Argus, if objects of a type are to be passed in messages, the type must be defined to be *transmissible.* Transmissible types have a canonical external representation (xrep) for objects, and user-defined *encode* and *decode* operations which translate from the internal representation to the xrep and vice versa, respectively [9]. When an object is passed as an argument in a remote invocation, Argus automatically invokes the encode operation before transmission, and decode upon receipt of the object.

2.4 An Example

To illustrate the form of Argus programs, we include here an example of a simple mail system based on a program in [12]. We have changed the program structure somewhat to hide more details of the mail system structure, so that it can be used to illustrate a wider class of possible modifications. This example will be referred to throughout the thesis in discussing various kinds of replacements to be performed.

The mail system consists of three guardian types: mailers, registries, and maildrops. Mailers provide the user interface to the mail system; maildrops store the messages for each user; registries hold the directory indicating which maildrop holds each user's messages. The module dependencies between these types are shown in Figure 2-2. Multiple instances of all three types are possible in any single mail system instance. Since the registries hold replicated information, using multiple registries will provide higher availability: if one registry crashes, the directory can be accessed from another registry. Multiple maildrops will reduce contention for service. A sample mail system configuration is shown in Figure 2-3. Clients of the mail system see only the handlers provided by mailer guardians. Registry and maildrop guardians are used only by the mailer. The user interface thus consists of read_mail, send_mail, add_user, and add_mailer handlers. In addition, a new instance of a mail system is instantiated via a call to the mailer$create operation.

Each user has a user_id. To read mail, the user calls the read_mail handler with the user_id as an argument. A list of messages is returned. Those messages are removed from the mail system state upon reply, so the next read_mail invoked by the user will not return messages that have already been seen. To send mail to another user, the send_mail handler is invoked giving the recipient's user_id, and the message, as arguments. To be added to the system, a user calls add_user and supplies a user_id. New mailers can be added to an existing mail system by invoking the add_mailer

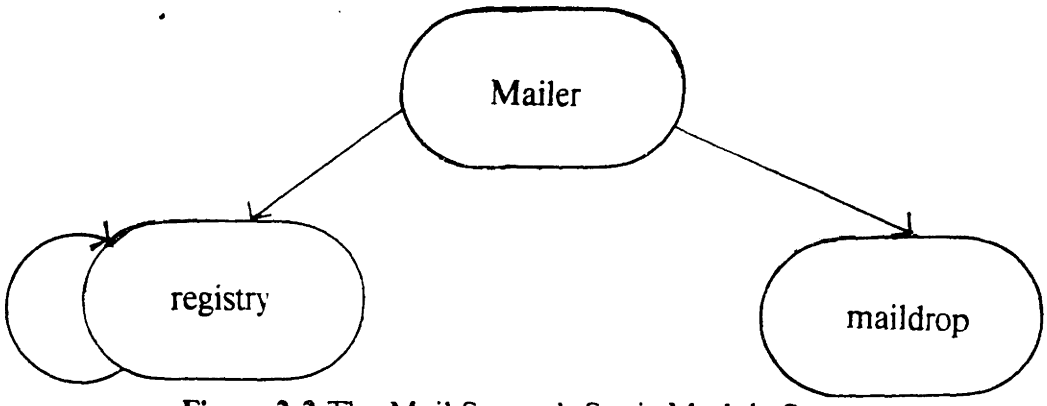

Figure 2-2: The Mail System's Static Module Structure

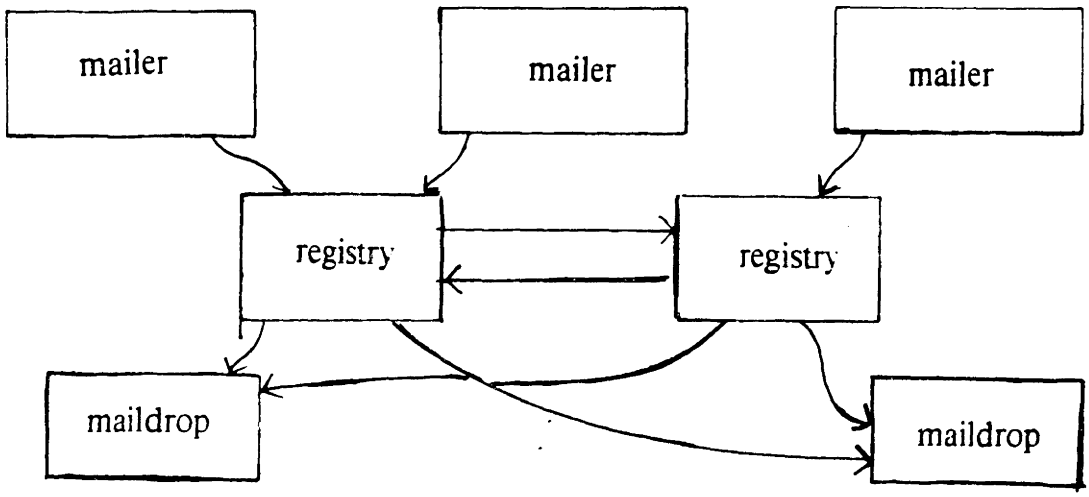

Figure 2-3: Example of a Mail System Instance

handler. Note that all of these invocations occur as subactions of the caller, and hence their effects will not be visible to other actions until the calling action commits.

The mailer also has several internal handlers, which can be invoked only from within the mailer guardian, as well as an internal creator, and a background process. The primary function of the background process is to determine when new maildrop and registry guardians are needed, and create the new instances of those guardian types. (We have not specified the algorithm used to make this determination, since

we are concerned here with the module structure of the mail system rather than with the procedures used.)

The internal creator, *new*, is used only by the add_mailer handler. *New* differs from *create* in that it receives as an argument an existing registry, whereas *create* instantiates a new registry. A mailer instantiated via *new* is joining an existing mail system and will have access to all existing registries and maildrops.

The internal handlers are add_registry and add_maildrop. These are used by the background process to add new components to the mail system. Note that the only handlers in the mailer that have registries or maildrops as arguments or return values are the internal handlers. Thus, these two component types are invisible to clients of the mail system; this structure will permit other implementations of the mail system to use different component types.

The maildrop guardians hold the users' mailboxes: it is here that messages are actually stored. Each maildrop has mailboxes for some subset of users; there is exactly one mailbox for each user in the system. The stable variable *boxes* in each maildrop points to that maildrop's list of mailboxes. Add_user, read_mail, and send_mail alter that list.

The registry guardian serves as a directory of mailbox locations, as well as a directory of the registries in the mail system. The registry state is replicated; each registry contains the list of all users, the location of each user's mailbox, and the list of all registries.

Registries, like mailers, have two creators, one of which is internal. *Create* is used by the first mailer to instantiate the first registry in the mail system; *new* is used to instantiate a registry with the same state as existing registries.

The *lookup* handler is used to locate the maildrop at which a given user's mailbox is located. *Select* picks a maildrop at which to place a new mailbox; however, it is the mailer, and not the registry, which informs the maildrop to add the user. *Add_registries* returns to the caller a list of all the current registries. The mailer uses that list to notify all registries of updates, such as the addition of a user, a maildrop, or a registry. This notification is accomplished by invocation of the registries' add_user, add_maildrop, and add_registry commands, respectively. In all cases where the mailer uses one of these commands, the command is invoked on all registries in parallel, within an atomic action, so that all of the registries will be updated, or none of them will be. The mailer thus ensures that the states of all the registries are kept consistent with one another.

The program text for the mail system is shown in Figure 2-5.

2.5 Library and Catalog

The Argus library maintains information about all user-defined abstractions in the Argus system. Each abstraction is represented in the library by a description unit (DU).

A DU contains a number of components. Here we discuss only guardian DU's. An example of a DU for the mail system is shown in Figure 2-4. The first component is the type specification for the abstraction. For guardians, the type specification is the set of handlers and creators. The DU also contains all of the *implementations* of the type. An implementation includes the source text of the module, as well as the compiled code for those processors on which the module can be run. Associated with each implementation is an *association list* that defines the bindings between external references to abstractions used in the implementation and the DU's for

those abstractions. DU's for guardians also contain a set of *guardian images.* The guardian image contains all of the information needed to construct the load module for the guardian: all of the code for the guardian, including the code for externally defined data types and procedures, and bindings to the DU's for the referenced guardian types.

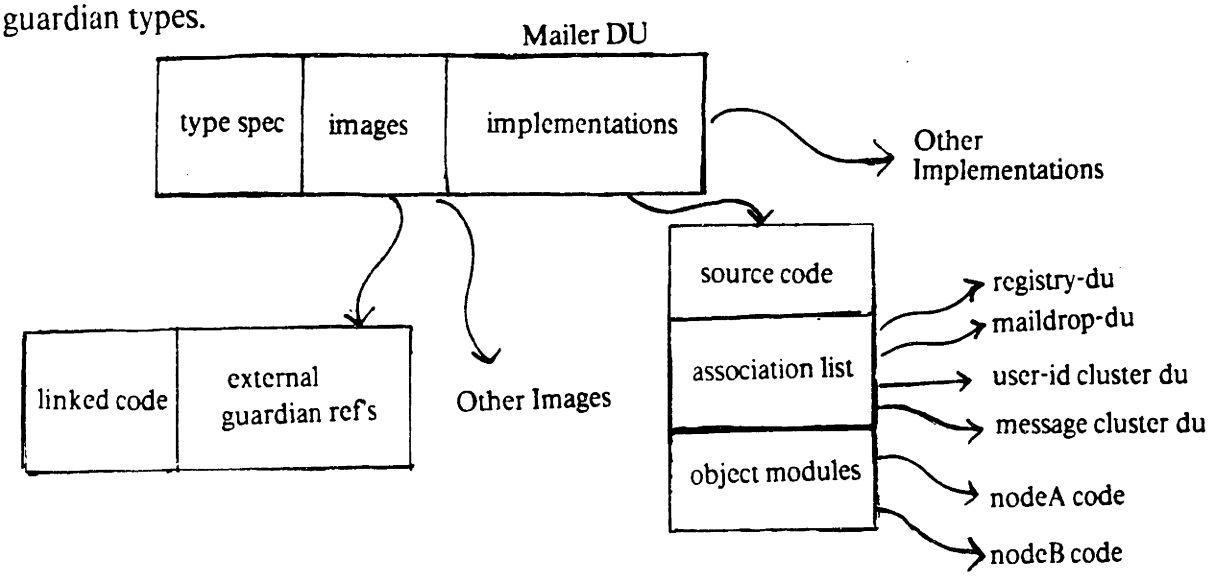

Figure 2-4:A DU for the Mailer

The catalog provides a directory of currently available instances of services. A user can provide a service name (usually the DU name) to the catalog and obtain a list of the current instances of that service. For our mail system, the mailer guardians would be listed in the catalog.

2.6 The Significance for Dynamic Replacement

There are a number of features of Argus that are relevant to our analysis of dynamic replacement. First, the fact that Argus modules have explicit interfaces, and a clear separation of interface from implementation makes dynamic, as well as static, modifiability considerably easier. The fact that only the processes within a guardian

can directly access its state, and that all clients communicate with a guardian only through handler calls makes it considerably easier to isolate guardians while changes are being made. Of course, the fact that the state structure we wish to maintain across replacements is completely implementation-dependent will make the task of moving that state value to a new implementation more difficult.

Argus' support of atomicity has an important impact on dynamic replacement. On one hand, it increases the constraints on the system. We now guarantee that permanent state will remain consistent, even across crashes and communications failures. The effects of committed actions will never be lost. Thus, a replacement facility must ensure that the most recent committed state of a guardian be carried over to any new version. On the other hand, this transaction facility ensures that the copy of the permanent state that resides on stable storage is always consistent, and is the only state to survive crashes. Thus, a guardian is prepared to crash at any time, and we know exactly what state will survive that crash. A replacement must only preserve the state that would normally survive crashes. Furthermore, we have a way to access a consistent copy of that state easily, since a mechanism must exist for retrieving the stable copy during crash recovery, and we can make use of that mechanism. Finally, it is important that clients are prepared for crashes of guardians that they use and will not automatically fail if a service becomes temporarily unavailable.

Finally, the location independence of inter-guardian communication provides the ability to redirect handler invocations to different guardians, and even to different nodes in the network, without having to notify the caller.

In summary, the kind of dynamic replacement we examine in this thesis is closely tied to the language for which we are considering a mechanism. Some of the relevant features we mentioned above, such as program modularization and clear

separation of interface from implementation are becoming more common in many languages. Other features, such as Argus' support of atomic actions, are still new to language designs, but greatly simplify replacement.

Figure 2-5: The Mail System Example

Mailer Guardian

```
mailer = guardian is create
                 handles send_mail, read_mail, add_user, add_mailer

reg_list = atomic_array[registry]          %type abbreviations
msg_list = atomic_array[message]

stable some: registry                      % stable state
best: registry                             % volatile state

recover
   best := some                            % reassign after crash
   end

background
   while true do                           %run this top-level action periodically
      enter topaction
         % decide if new registry or maildrop is needed
         % if yes, call add_registry, add_maildrop
         % select a new best registry, e.g. closest responding
         end
      sleep(...)
      end
   end

create = creator() returns (mailer)        %creates new instance
   some := registry$create()               %initialize state
   best := some
   return(self)                            %self names interface object for this guardian
   end create

new = creator(reg:registry) returns (mailer)
   some:= reg                              %initializes state to point to existing registry.
   best:= reg
   return(self)
   end new
```

```
read_mail = handler (user: user_id) returns (msg_list)
                    signals (no_such_user)
                                                    % find maildrop; invoke its read_mail handler;
    return(best.lookup(user).read_mail(user))       %return messages
      resignal no_such_user                         %unless user doesn't exist
    end read_mail

send_mail = handler (user: user_id, msg: message) signals (no_such_user)
    best.lookup(user).send_mail(user, msg)          % send message to maildrop
      resignal no_such_user                         %user doesn't exist
    end send_mail

add_user = handler (user: user_id) signals (user_exists)
    drop: maildrop := best.select()                 %pick a maildrop
      resignal user_exists
    all: reg_list := best.all_registries()          %get all registries
    coenter                                         %in parallel,
      action                                        %add user to maildrop,
          drop.add_user(user)
      action foreach reg: registry in reg_list$elements(all)
          reg.add_user(user, drop)                  % and to each registry.
      end
    end add_user

add_mailer = handler(home: node) returns (mailer)
    return(mailer$new(best)@home)                   %calls internal creator
    end add_mailer

add_maildrop = handler (home:node)
    drop: maildrop := maildrop$create()@home        %create the maildrop
    all:reg_list := best.all_registries()
    coenter action foreach reg: registry in reg_list$elements(all)
        reg.add_maildrop(drop)                      %and inform all registries.
        end
    end add_maildrop

add_registry = handler ()
    new: registry := best.new_registry(home)
    all:reg_list := best.all_registries()
    coenter action foreach reg: registry in reg_list$elements(all)
        reg.add_registry(new)
        end
    end add_registry
end mailer
```

Figure 2-5: (continued)

Registry Guardian

```
registry = guardian is create
                handles lookup, select, all_registries,
                        add_user, add_maildrop, add_registry, new_registry

reg_list = atomic_array[registry]                          %type abbreviations
steer_list = atomic_array[steering]
steering = struct[users:user_list, drop:maildrop]
user_list = atomic_array[user_id]

stable regs: reg_list    % stable state: all registries and
stable steers: steer_list % all users and maildrops

create = creator () returns (registry)
    regs: = reg_list$new()                                 %initialize state
    steers : = steer_list$new()
    reg_list$addh(regs, self)                              %add self to list of regs
    return(self)
    end create

new = creator(rlist:reg_list, slist:steer_list)%internal creator
    reg_list$addh(rlist, self)              %add self to list of regs
    regs : = rlist      %initialize state
    steers : = slist
    return(self)
    end new

lookup = handler (user: user_id) returns (maildrop)signals (no_such_user)
    for steer: steering in steer_list$elements(steers) do
        for usr: user_id in user_list$elements(steer.users) do
            if usr = user then return(steer.drop) end
            end
        end
    signal no_such_user
    end lookup
```

Figure 2-5: (continued)

```
select = handler (user:user_id) returns (maildrop) signals (user_exists)
    for steer:steering in steer_list$elements(steers) do
        for usr:user_id in user_list$elements(steer.users) do
            if usr = user then signal user_exists end
            end
        end
    drop: maildrop := ... % choose, e.g., maildrop with least users
    return(drop)
    end select

all_registries = handler () returns (reg_list)
    return(regs)
    end all_registries

add_user = handler (user: user_id, drop: maildrop) signals (user_exists)
    for steer: steering in steer_list$elements(steers) do
        if steer.drop = drop
            then user_list$addh(steer.users, user)          %append user
        end
    end add_user

add_maildrop = handler (drop: maildrop)
    steer_list$addh(steers, steering${users: user_list$new(), drop: drop})
    end add_maildrop

add_registry = handler (reg: registry)
    reg_list$addh(registries, reg)                  %adds new registry to list in state
    end add_registry

new_registry = handler(home:node) returns (registry)
    return(registry$new(regs, steers)@home)         %calls internal creator
    end new_registry

end registry
```

Figure 2-5: (continued)

```
Maildrop Guardian

maildrop = guardian is create
           handles send_mail, read_mail, add_user

box_list = atomic_array[mailbox]
mailbox = struct[mail: msg_list, user:user_id]
msg_list = atomic_array[message]

stable boxes: box_list := box_list$new()

create = creator () returns (maildrop)
   return(self)
   end create

send_mail = handler (user: user_id, msg: message) signals(no_such_user)
   for box: mailbox in box_list$elements(boxes) do
      if box.user = user
         then msg_list$addh(box.mail, msg)                    append message
              return
         end
      end
   end send_mail

read_mail = handler (user: user_id) returns (msg_list)signal (no_such_user)

   %search for user's mailbox
   for box: mailbox in box_list$elements(boxes) do
      if box.user = user                                      %when found, copy messages
         then mail: msg_list := msg_list$copy(box.mail)
              msg_list$trim(box.mail, 1, 0)                   % delete messages from box
              return(mail)
         end
      end
   end read_mail

add_user = handler (user: user_id)
   box_list$addh(boxes, mailbox${mail: msg_list$new(),user: user})   % add new box to state
   end add_user

end maildrop
```

Figure 2-5: (continued)

Chapter Three

The Unit of Replacement

In the previous chapter we described the structure of Argus programs; we must now determine how dynamic replacement should be used for such structures. Argus programs contain several kinds of modules: clusters, procedures, iterators, and guardians. Dynamic replacement could be supported for any of these module types and conceivably for parts of those modules as well, for example, individual cluster operations or handlers. In addition, groups of these modules could be replaced together. In this chapter, we discuss the Argus program units for which we intend to support dynamic replacement. The reasons for our choice are presented. We then describe the information about those units that is required by a replacement mechanism to guarantee preservation of the module's interface to clients across replacement.

3.1 Selecting the Unit of Replacement

In this section, we determine which Argus module types will be replaceable. While the basic structuring unit for Argus programs is the guardian, we have two options other than guardians as possible units of replacement. First, guardians contain procedure and cluster implementations, as well as the guardian components, such as handlers and creators. We could choose to make any of these module types independently replaceable. Conversely, there are abstractions that are composed of sets of guardians that cooperate to provide a single service, such as the mail system.

We have determined that guardians should be considered the minimal units of

replacement in Argus. There are several reasons for this choice. First, guardians have a small, well-defined interface to the rest of the system, described by the set of handlers provided. A guardian forms a stand-alone unit that contains all of the code that directly accesses any of its state, and (logically) has its own copies of all data type and procedure implementations. There is thus a strong boundary between a guardian and the rest of the system that allows the replacement of a single guardian instance without affecting the rest of the system or even other instances of the same guardian type, whether at the same or different nodes in the network. This structure makes it easier to replace the guardian without affecting the rest of the system. Second, guardians have an explicit permanent state. Since a guardian is expected to resume execution from that state alone after a crash, it is not unreasonable to assume the same after a replacement. Thus, a firm limit exists on the state that must be moved to the new version during a replacement.

We have decided against supporting the replacement of units smaller than guardians for the following reasons. Though a cluster may be used by many different guardians, each of those guardians may use a different implementation of the same cluster; however only one implementation will exist per guardian. Hence, if a modification is being made to one implementation, replacement will involve locating the guardians which use that cluster implementation, and replacing the code (and possibly state) of those guardians. Replacing the implementation of a cluster or procedure can therefore be considered a special case of replacing the entire guardian. Techniques we develop for guardian replacement will apply. If greater efficiency is sought in replacing these smaller modules, techniques similar to those developed in [5, 7] can be used.

Finally, even if smaller units of replacement were permitted, the replacement of guardians as single units would still be necessary. This requirement stems from the sharing of guardian state among all the guardian's handlers. If the structure of the

permanent state changes, all handlers that access the state must be updated at the same time the state is changed. Thus, the entire guardian must be replaced indivisibly with respect to handler usage. We therefore will not support replacement of units smaller than guardians.

The question now arises of whether we wish to provide a mechanism to replace units larger than guardians. While sets of guardians can cooperate to provide a single service to clients, the language does not recognize the connection among those guardians. Thus, while the mail system described in the previous chapter is a service composed of mailer, registry, and maildrop guardians, there is no way to explicitly state that those guardians are part of the implementation of a single abstraction. We refer to these multi-guardian implementations as *subsystems.*

The ability to replace individual guardians does not provide the power to replace a subsystem without restricting the replacement to preserving some of the implementation details of the subsystem along with preserving the interface. When some of the handlers provided by component guardians are used only by other components of the subsystem instance, their use is an implementation decision; other implementations of the subsystem might use other types of components that provide different handlers. Such implementation decisions should be subject to change during dynamic replacement. If each guardian is replaced individually, its entire interface must be preserved across the replacement, since there is no way to determine that the only guardians using some of its handlers are being replaced at the same time. Hence, there is a difference between replacing the set of guardians comprising a subsystem (requiring that all of their handlers be preserved) and replacing the subsystem as a single entity. (It is also possible that some of a given component's handlers are available to clients, while other handlers are used only for communication within the subsystem. In such cases, the handlers visible to clients must be preserved, but the component type as a whole need not be.)

The mail system (Figure 2-5), is an example of a subsystem in which some components are not visible in the interface. Only the mailer guardians are visible to clients of the system; registries and maildrops are not. Thus, the same service could have been implemented with a different set of underlying guardian types. The difference would not be apparent to clients. In the implementation shown here, the mailer abstraction is identical to the mail system abstraction; the mailer provides the client interface to the mail system. In other cases, there is no single guardian providing the entire subsystem interface.

Unfortunately, Argus cannot distinguish between guardians that are used solely within a subsystem and those that are available to clients. To Argus, there is no difference between the mailer guardian and the registry guardian in our mail system example. The lack of any distinction between visible and hidden guardians in a subsystem presents a dilemma for the replacement mechanism. One of the requirements placed on the replacement system is that it support the methodology associated with CLU and Argus. Thus, it is expected that in keeping with the principle of strong typing, the replacement mechanism will allow replacements only between two implementations of the same type. If there is no way to identify those guardians that do not contribute to the interface, the replacement system has no choice but to require that each individual guardian interface be preserved.

On the other hand, the principle of separating interface from implementation dictates that changes in the implementation of an abstraction should be invisible to clients, and therefore implementation details should be subject to modification. Component guardian types are implementation decisions. Since registries and maildrops do not contribute to the interface, the replacement system should not require that they be preserved across replacements.

Hence, the replacement system will have to support replacement of subsystems as

single entities, if it is to ensure type safety without imposing undue restrictions on the kinds of replacement permitted.

Some explicit support for subsystems will be needed in the system, so that the replacement mechanism can distinguish between those handlers that must be preserved across replacement and those that may be eliminated. At this point, we will not consider adding a subsystem abstraction mechanism to the language as a method of maintaining this information. Adding another module type to Argus adds significant complexity. Before such an option can be adopted, the need for subsystems for uses other than replacement must be explored. The importance of subsystem entities must be weighed against the need for simplicity in the language. Instead, we will define subsystems in the Argus library, and maintain the needed information about subsystem interfaces there.

In the remainder of this chapter we will discuss the forms in which subsystems appear in Argus programs, what information about those subsystem structures must be maintained for use by the replacement system, and how that information can be incorporated in the Argus library.

3.2 Subsystem Structures in Argus

Intuitively, a subsystem is any collection of modules that provides a unified interface for its clients. The interface can be a set of operations, like a cluster interface, or a set of handlers, like a guardian interface.

As with guardians, subsystems can have long-term state that survives crashes. While not all subsystems have robust, long-term state, we are concerned here with those that do, since the problem of dynamic replacement arises primarily in those cases. Hence, we will limit the discussion here to subsystems having long-term state. This

restriction implies that we will not consider subsystems composed solely of clusters since clusters do not have their own stable storage; we will consider only those subsystems composed of guardians or a combination of clusters and guardians. Although a guardian is the simplest form of a subsystem, for the remainder of this section we will discuss only multi-module subsystems, since single guardians need no additional support.

We first clarify our terminology. The term *subsystem* refers to an active, executable module in the system, while the term *subsystem definition* refers to the textual definition of the modules used in a subsystem instance. We will use the term *subsystem instance* to mean subsystem, when we wish to emphasize the distinction between the active entity and its definition. At any given time, there may be many subsystem instances associated with the same subsystem definition.

Unlike individual guardians, subsystems may span nodes in the system. In addition, since subsystems may be composed of many guardians, they can be built up using previously defined subsystem and guardian types. Guardians cannot contain other guardians. Subsystems are thus hierarchically structured in a way that guardians are not[2]. One benefit that arises from the greater flexibility of subsystem structures is that a subsystem may have many handlers of a given type, while a guardian can have no more than one. In subsystems, multiple handlers of the same type occur whenever many guardians of the same type are components of the same subsystem instance. In the mail system example, there will probably be one mailer at each node and therefore many handlers of each type provided by the subsystem. Another use of this structure is to maintain separate connections with each client: a

---

[2]By hierarchically structured we mean only that subsystems can be built from many other previously defined subsystem modules. Argus does not provide any block structuring or nesting of module definitions.

new guardian instance is created each time a new client requires service. That guardian instance can terminate when the "conversation" with its client ends.

In the remainder of this section, we will discuss the various ways in which subsystems can be represented in Argus, i.e. the structures that can be used in Argus programs to provide a unified interface for a set of modules. We start with subsystems composed solely of guardians, which we call guardian-based subsystems, and then discuss the reasons why guardians alone may be considered insufficient for creating the desired structures, and how clusters are used in these cases. Subsystems composed of combinations of guardians and clusters are referred to as cluster-based, since, as will be evident later, the clusters appear at the "root" of the subsystem.

For each subsystem form, we examine the additional information about the psubsystem that the replacement system will need to know to distinguish between the subsystem interface and the component guardians' interfaces. We also examine ways in which that information can be derived and maintained. It should be emphasized that while this classification of subsystem structures is useful for analyzing the relationship between the structure and its logical interface, there is no way to distinguish among the various forms from syntactic structure.

3.2.1 Guardian-Based Subsystems with Centralized Interfaces

The most straightforward module structure for a subsystem is that exhibited by the mail system. The subsystem interface is just the interface to a single guardian type (here, the mailer). The static module structure for the mail system was shown in Figure 2-2. In this structure, we can clearly identify the *root* of the subsystem, i.e. one guardian type that is the root of the subsystem definition's tree structure (i.e. the mailer guardian). All communication between client and subsystem instances pass through a guardian of the root type. Only the root guardians are visible to clients; all other components may be unique to a particular implementation.

Note that there is a difference between the static module structure, and the structure of a subsystem instance. An example of a possible configuration for an instance of the mail system was shown in Figure 2-3. Any number of instances of the root type may exist, possibly connected to different sets of clients.

If all subsystems had this form, the only information we would need is the identity of the root guardian type, and the types of the component guardians. With this information, the replacement system could safely allow component guardians to be omitted from an instantiation of a new implementation, while ensuring that the root (and hence the interface) was preserved.

The Argus library has a description unit (DU) for each abstraction defined in the language (see section 2.5). The necessary subsystem type information can be kept in the DU for the root guardian type. For each implementation in the DU, we must add a list of the types of all of the component guardians in the subsystem. We therefore have a new declaration associated with each implementation in the root's DU:

includes component_type$_1$, ..., component_type$_n$

For the mail system implementation in Figure 2-5, we would have the following includes clause associated with the implementation in the mailer DU:

includes registry, maildrop

With this information in the mailer DU, the replacement system can determine that the mailer interface must be preserved, but that registries and maildrops may be eliminated by a replacement.

3.2.2 Guardian-Based Subsystems with Distributed Interfaces

Since other forms of subsystems exist in addition to the simple structure described above, more complex support is required. It is possible for subsystem interfaces to include, not only the handlers from the root guardian, but handlers from component guardians as well. There are three ways such a distributed interface can be provided to users. One is to pass the handlers from component guardians to clients in reply to invocations of the root's handlers. The second is to store those handlers in the catalog (see section 2.5). The third is to construct an explicit subsystem interface type, store the collection of handlers in an object of this type, and return that object in response to invocation of the root's creator(s).

A simple example of a subsystem with a distributed interface is a slightly different mail system in which the add_user handler returned the read_mail handler from the maildrop at which the user's mailbox was located. The user interface to the mail system would then consist of the mailer's handlers plus one maildrop handler.

Several issues arise in attempting to maintain adequate interface information for these kinds of subsystem structures. First, unlike the previous case, the interface is no longer derivable from the root guardian's interface. To explain why, we must first define the *closure* of the interface. The closure of a guardian interface is defined to be the set of handlers reachable from the object(s) returned from the creator of the guardian, where reachable means the handler is contained in the object, or can be obtained as the result of invoking a handler in that object, or can be obtained as the result of invoking another handler reachable from the object.[3] In

---

[3] If there is more than one creator, the closure is all the handlers reachable from all of the creators. This definition is somewhat restrictive in that a given user will most likely have access to the guardian through one creator only. However, we will be unable to distinguish among different interfaces to a single guardian.

most cases, the interface will be the set of handlers in the *closure* under handler invocation of the handlers in the root guardian interface. There are, however, cases in which the subsystem interface is not the set of handlers in the closure of the root interface. One case occurs when a handler that is passed to clients in reply to an invocation is not a handler of the subsystem; in this case, there is a handler in the interface closure that is not conceptually part of the subsystem interface. The second exception occurs when, rather than returning a component's handler in reply to an invocation, the root (or the component itself) stores the handler in the catalog, to be retrieved from there by clients. In this case the closure of the root interface is only a subset of the subsystem interface. Hence, we cannot automatically determine the subsystem interface from the root's interface.

An explicit declaration of the interface will be required in the root DU, along with the component list. We therefore add the following clause to the DU interface definition:

subsys_handles <handler_list>

where handler_list is a list of the handlers in the subsystem interface. Note that this clause is needed only once per DU, not once per implementation as for *includes* clauses. For the modified mail system described above, the clause would look like:

subsys_handles add_user, read_mail, send_mail, add_mailer

The replacement system will have to depend upon the correct specification of this handler list: it should include those handlers provided through the catalog, and those provided through invocation of other handlers, as well as those in the root's *handles* list.

The *subsys_handles* clause is distinct from the *handles* clause in the mailer that defines the handler interface to the root alone. Ideally, we would like a single handles list in the DU that defines the subsystem interface. We would then require

that every handler listed be provided by either the root or some other component of the subsystem. Semantically, there should be no distinction between the handlers provided by the root and those provided by other components. We cannot eliminate the syntactic distinction here because the change would involve changing Argus type definitions. A guardian definition is expected to contain handler definitions for each of the handlers named in the guardian's *handles* list.

Another problem that arises from use of distributed interfaces is illustrated by the example given earlier in which the add_user command returned the read_mail handler for the maildrop at which that user was placed. The mailer interface still contains a read_mail handler; the client is free to use either. The type specification for a handler contains the handler's name (i.e. read_mail), as well as the number and types of its arguments and return values. Hence, the type of both read_mail handlers is the same. The fact that they derive from different guardians does not affect the type. This situation does not pose a problem for the Argus typechecker. From the client's point of view, both handlers yield the same result. From the point of view of a dynamic replacement mechanism however, it presents a problem. The replacement system must know which handlers are in the interface because it must know which must be preserved across replacements. If only the read_mail handler of the mailer is in the interface, then the maildrops may be eliminated during a replacement. If it is the maildrops' handlers that are in the interface, the replacement system cannot allow those handlers to disappear.

Thus, still more information must be maintained for replacement: both the explicit type specification for the subsystem (i.e. the set of handler type specifications), and information as to which component guardian provides each of those handlers. Obviously, more than one component type can be listed for each handler type, and vice versa. The *includes* list must be changed to:

includes component_type$_1$ ({handlers}),

....

component_type$_n$({handlers})

The component_types are the same as those described in the simpler *includes* list. The set {handler names} is zero or more handlers from the interface of corresponding guardian type that are in the subsystem interface. Every handler specified in the subsys_handles list must be provided by the root guardian or listed in *includes* list. A new mailer structure with the change discussed earlier would have the following includes list:

includes maildrop(read_mail)

This extended form of the includes clause, along with the subsys_handles interface specification will enable the replacement system to distinguish between the subsystem interface and its implementation during replacement.

3.2.3 Cluster-Based Subsystems

We now examine those subsystems for which the implementation is a combination of guardians and clusters. Cluster-based subsystems are distinguished from guardian-based systems in that at least some of the operations in the interface are cluster operations, rather than handlers or guardian creators. (Syntactically, the two forms of subsystems may be indistinguishable to clients, for example, if the only cluster operation is a creation operation.) In Argus, all clusters and data objects reside in guardian instances: the Argus universe is partitioned into guardian address spaces. When we say subsystems are composed of both guardians and clusters, we mean that there are clusters that function as part of the subsystem, but are instantiated outside the guardian components of the subsystem. Instead, they are

instantiated in the guardians that are clients of the subsystem. The structure then looks like that in Figure 3-1.

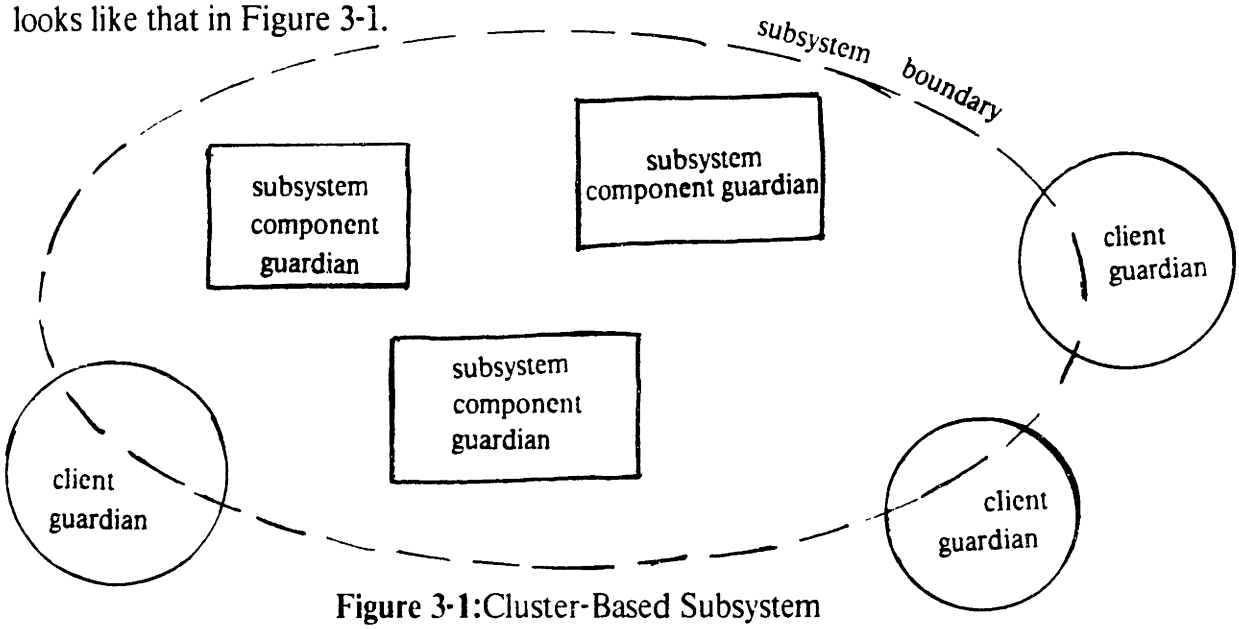

Figure 3-1: Cluster-Based Subsystem

Some functions that are logically part of the subsystem are thus located at client sites. In guardian-based subsystems, the only subsystem data at client sites are the handler identifiers.

The simplest kind of cluster-based subsystem is one in which the cluster manages creation of the subsystem's components, but performs no other function except providing the handlers. Instead of creating all of the component guardians for a subsystem in the root guardian and returning the distributed interface in response to a creation of that guardian (as in our previous example), the subsystem could be structured so that all of the components are created in the cluster that serves as the interface type. We could for example, have a mailsys cluster whose representation is

```
struct[read_mail:handler(user_id) returns (msg_list),
       send_mail:handler(user_id, msg),
       add_user:handler(user_id),
       add_mailer:handler() returns (mailsys)]
```

The create operation of the cluster would create both the mailer and registry guardians and return the handlers for the mail system. This structure would eliminate the need for an internal creator in the mailer. Since the cluster creates the registries also, a single creator that takes a registry as an argument is sufficient. The other operations of the cluster are get_*handler* operations that can be used with the "." syntactic sugar, as in mailsys.read_mail, for example.

In other cluster-based subsystems, the entire interface is a set of cluster operations. For example, we could have constructed the mail system with the mailer as a cluster instead of as a guardian. The registries and maildrops would exist as before. The function of assigning new users to registries and forwarding read_mail and send_mail requests would be handled by the mailer cluster. However, the access to a registry, now stored in the mailer's permanent state, would have to be stored instead in the catalog (unless it could be ensured that every guardian containing a mailer cluster stored the mailer's state in its stable storage.) The cluster could store this knowledge in volatile state as part of its rep. Only in the event of a crash would it have to access the catalog.

This structure might be used to increase efficiency of the subsystem: clusters are assumed to be cheaper than guardians, and procedure calls are cheaper than inter-guardian communication. However, the extra overhead attributable to separating the subsystem implementation from the client cannot be accurately estimated without more extensive performance evaluation. Even if there are gains in efficiency from using a cluster-based structure, there is a clear loss of modularity in replicating a function at each site of use. Since there is a well-structured way to implement these subsystems as collections of guardians, we do not think it necessary to provide special support for replacement of this kind of cluster-based structure. It will be possible to replace an entire subsystem of this form only by replacing all client guardians along with the subsystem. While it is a goal of this work to support

replacement of as wide a range of subsystems as possible, modifiability is always dependent on the modularity of the program. There is a tradeoff here between efficiency and modifiability, and we will be unable to support the full range of dynamic modifications for subsystems structured as described.

Limited support for these cluster-based structures can be provided in the library without significant additional mechanism. The cluster can be considered the "root" of the subsystem, and the subsystem information can be included in that cluster's DU in the library. The *subsys_provides* clause can be permitted in cluster definitions as well as guardian definitions, so they may be attached to these interface type definitions. *Includes* clauses can be associated with different implementations of the root cluster, as they are with guardian roots. Replacement will be possible when installing changes that do not modify the cluster objects at client sites.

From these descriptions it is apparent that cluster-based subsystems vary considerably in structure. In particular, parts of the function and data of the subsystem may be distributed and replicated among clients. In such cases, we lose the clear boundary of the subsystem afforded by guardians. It becomes difficult or impossible to replace the subsystem implementation without locating and modifying all clients as well. For the remainder of this thesis, we will limit discussion to guardian-based subsystems. For those cases of cluster-based subsystems in which the objects of the cluster do not change value, our results will apply.

3.3 Recognizing Subsystem Instances

Thus far, we have discussed the type information that must be maintained to allow the replacement system to type check subsystems adequately during replacement. This information relates the subsystem instances in existence to DU's in the library. We now examine a different aspect of the problem.

Many instances of a given subsystem type may exist at any one time. The replacement system has no information as to which component guardians belong to which subsystem instances. Thus, if the user wishes to replace a single subsystem instance, the replacement system has no way to locate the components that comprise that instance: it must be left to the user to identify each component explicitly. If the user mistakenly names a component from another subsystem instance, or omits one from this instance, the replacement system cannot detect the error. In this section, we examine the facility that would be required to keep track of subsystem instances and the guardian components belonging to each.

Subsystem structures are not static; guardians can be added or removed from the subsystem throughout its lifetime. Thus, to maintain the information on the components of a subsystem instance not only requires knowledge of the creation of the instance, but knowledge of the creation of components within the instance. Unlike the interface specifications, the instance information cannot be declared by the programmer; it must be generated during execution. There are a number of reasons that such knowledge is difficult to cull automatically from Argus programs.

One problem is recognizing the creation of subsystem components. Guardians often create other guardians that are not necessarily part of the same subsystem. We therefore have no way to determine automatically whether a guardian creation implies addition of a component to the subsystem. Since, the library has the list of component types for the subsystem, we could make the judgment that if the guardian created from within a subsystem is of a component type (or in the closure of the component list), then it must be a new subsystem component, and otherwise, the new guardian is assumed to be independent. However, this rule is not completely accurate either. A guardian type might be used both as a subsystem component and independently. Though it is unlikely that such a structure would occur, it is feasible, and therefore must be taken into account.

The only way to maintain subsystem instance information accurately is with language support. We would not only have to recognize subsystems in the language, but also distinguish between component creation and independent guardian creation. We would need two different kinds of guardian creator invocations: one to create a guardian in the invoker's subsystem and one to create the guardian independently. The inclusion of such operations would be a major change in the language semantics, since it introduces the notion of subsystem explicitly.

Because it is not possible to accurately and easily retain the subsystem instance information needed, we have decided not to maintain the information during execution. Instead, as part of the replacement mechanism, we will provide support for users of the replacement system to locate the components that comprise a given subsystem instance at the time of replacement (Section 6.2). This support will make it easier for users to identify the components of a subsystem instance correctly. However, we will be unable to check that the guardians identified by the user belong to the same subsystem instance, or that all components of the instance have been found.

3.4 Conclusions

From this discussion of subsystem support in Argus, we can draw a number of conclusions. Our primary reason for examining subsystems in the context of replacement was that we were faced with a choice between flexibility and safety: without identifying subsystems explicitly the replacement system would have to either allow modifications that were unsafe, or disallow certain changes that should have been possible. We have attempted to compromise by including support for subsystems in the library without changing the language. That compromise has been partially successful. It provides the information the replacement system needs

to distinguish between the subsystem interface and its component interfaces. The replacement system now has more information upon which to depend in deciding whether a modification to subsystem structure is valid. However, we will support only subsystems that are constructed as sets of guardians; we will not provide special support for subsystems that have component clusters at client sites.

There are still weaknesses in the type safety of subsystems. However, those weaknesses appear to be cases of weaknesses already existing in the language; for example it has always been possible to "violate the abstraction" in CLU and Argus by passing an internal cluster operation or handler to clients. Such a type violation is analogous to passing to clients a handler that is not listed in the subsys_handles list. However, since the subsys_handles list is in the library only, rather than in the program text, there is probably a greater likelihood of unintentionally omitting a handler from that list than there is of unintentionally passing an internal operation out of a cluster. Thus, the possibility of user error, rather than intentional circumvention of the type mechanism, is greater when performing replacement than when constructing Argus programs. We also will be unable to guard against errors resulting from incorrectly identifying the set of component guardians in a given subsystem instance.

Chapter Four

Legality of Replacement

In this chapter, we establish criteria for safely replacing one subsystem instance with another. If the Argus system is to allow dynamic replacement without undermining the modularity and reliability of the system, we must ensure that modules can continue to depend on the correct functioning of the modules they use, regardless of whether replacements are performed on those modules.

Intuitively, if two modules satisfy the same interface specification, one can be used in place of the other. When a given abstraction is needed, any implementation of that abstraction can be used.[4] We have found that when dynamic replacement is possible, it becomes more difficult to determine exactly what this intuitive criterion means, and when it is satisfied. In some cases, one implementation of an Argus subsystem abstraction cannot be used to replace another implementation of the same abstraction. Conversely, an implementation of a different abstraction may behave appropriately if substituted for that running instance.

In this chapter, we first present some examples to illustrate the complexity of the issues, and then give an intuitive description of the problems in determining the legality of a replacement. We then present a more formal model of the system that will enable us to define safety criteria and to characterize more precisely the situations in which our initial intuitions about the safety of a replacement are likely

---

[4]We do not distinguish between single and multi-guardian subsystems. The ability to distinguish between the subsystem interface and its component interfaces, as described in the previous chapter, is assumed.

to be incorrect. Finally, the formal model will make apparent a number of issues concerning the effects of replacements that were not immediately obvious from the examples given. Our goal in providing a formal model is not to provide support for automatic verification of replacements, but rather to provide a basis on which implementors performing replacements can reason about the safety of various modifications.

We first clarify some basic terminology. When a subsystem instance is running in the Argus system, clients of that subsystem will see a single module whose execution may be interrupted by crashes, but whose identity never changes. When dynamic replacements occur, the code and state within the subsystem change, but its identity does not. In discussing replacement, we must clearly distinguish between the client's view and the implementor's view. Hence, we will use the term *module* throughout this chapter to refer to the continuously-existing subsystem object seen by clients, and the term *instance* to refer to the combination of code and state that is replaced. A module, over its lifetime, is thus comprised of a series of instances, each picking up the execution from the point at which the previous instance left off. We will give a more precise definition of instance later in the chapter.

The terms abstraction, implementation and specification are used as in the description of the Argus library in Section 2.5. Two implementations implement the same abstraction if and only if they both satisfy the specification of the abstraction. We will give more precise definitions of these terms later as well. We use the term *replacing instance* to refer to the instance to be installed in place of the currently running instance, which we refer to as the *replaced instance.*

4.1 An Intuitive Description of the Problem

There are two separate problems to be examined. One is whether one abstraction can be used to replace another, and the second is whether a given implementation of an abstraction can be used in the instance replacing a given running instance.

4.1.1 Abstraction Replacement

The simplest definition of replacement would require that the abstraction remain unchanged across replacements, i.e. that only implementations of an abstraction be changed during dynamic replacement. There are cases, however, in which this intuitive criterion can be relaxed, while ensuring that the behavior of the module does not change in any way visible to its clients. There are also cases in which satisfying the initial abstraction is not sufficient to ensure that a new instance can replace an existing instance of the same abstraction.

We are interested in relaxing the restriction in two cases. One is when a new abstraction is designed specifically to replace existing instances and will produce acceptable behavior in continuing from some non-initial state, but could not handle some cases that could have arisen prior to replacement. The second case is when the initial abstraction did not contain some operations later determined to be necessary. We would then like to replace the instance with an instance of an *extended abstraction* that will allow existing clients to continue their use of the module, while allowing newer clients to use the extended set of operations. In the remainder of this section we first discuss why requiring the same abstraction is insufficient, and when replacement abstractions with the same interface can be used. We then discuss the relationship that must hold between the replaced instance and the replacing instance. The extension problem will be addressed last.

To illustrate the situations that must be examined in determining criteria for legal

replacement, we consider an example of a unique-id generator. The specification of the abstraction states that no id will be repeated for at least X invocations of the next_id operation, for some large number X. This specification will allow any sequences of id's to be generated as long as there are no repetitions within the first X id's. Now suppose that after some number of id's, N, has been generated, we want to replace the existing instance with a newer implementation (perhaps one with a faster algorithm). What specification must the replacement satisfy?

Satisfying the original specification alone is not sufficient. Not only must the new instance not repeat the id's it generates, it must also not repeat the id's generated by its predecessor. Thus the set of allowable sequences of id's is now smaller than that allowed by the original specification.

On the other hand, clients depend on the module to produce only X id's, since that was the specification of the module when they originally connected to it. Thus, a replacing instance installed after the generation of N id's actually must guarantee only X-N unique-id's to satisfy the module's specification on which clients depend. Hence, in one respect, the specification of the *continuation abstraction* is stronger than the original, while in other respects it is weaker. Note that an implementation of the original abstraction might satisfy both the original specification and the continuation specification, but there could be implementations of the original abstraction that do not satisfy the continuation specification. (However, it must be true that any implementation of an abstraction can replace itself, since an instance could be stopped at any time and replaced with an exact copy of itself.)

It is also evident from this example that a replacing instance does not necessarily have to produce the same replies to future operations as the replaced instance would have, had it continued to run. The replacing instance of the unique-id generator need not produce the same id's in the same order as the replaced instance would

have. Since clients can depend only on the specification, the differences will not be observable if the replacing instance satisfies the continuation specification.

This example suggests several areas that must be considered in defining dynamic replacement. First, we need a way to derive continuation specifications from original module specifications. We must define what it means for an implementation to satisfy a continuation specification.

Second, the notions of abstraction and specification change when continuation abstractions are introduced. Without dynamic replacement, there was an implicit assumption that instances were always started in an "initial state" defined by the creator operations, and possibly the arguments and parameters to the creators. A continuation abstraction assumes that the instance is starting with some state defined in part by another instance, and in part by the replacement procedure. The specification of the abstraction's behavior depends on prior events having occurred. For example, if we replaced the mail system described in Figure 2-5 after msg1 had been sent to user1, it should be possible to deduce from the continuation specification that the next read_mail(user1) invocation would return msg1 (and possibly other messages as well). Rather than specifying the behavior of an abstraction from the time of creation, specifications must now describe the behavior of abstractions from a given state. In section 4.2, we will use a definition of abstraction that incorporates this notion. We will also discuss the implications of this notion for defining whether an implementation satisfies a specification, and whether a given implementation can be used in replacing a running instance.

4.1.2 Instance Replacement

If an instance of abstraction$_i$ is to be replaced with an instance of abstraction$_j$, there are correctness conditions on the instance of abstraction$_j$ that must be verified even

if it is known that abstraction$_j$ meets the correctness criteria for abstraction replacement. These conditions primarily involve the correspondence between the final state of the replaced instance and the starting state of the replacing instance.

The replacing instance must start in a state corresponding to the state in which the replaced instance ended. Since the state format itself is implementation dependent, we must have a notion of *preserving the value* of the state. For the mail system, that value is (intuitively) the set of users and their current mail. A new implementation of the mail system might not store that value in atomic_arrays in registry and maildrop guardians, but it must store enough information so that future read_mail and add_user operations, for example, will produce the same results as if a replacement had not taken place.

4.1.3 Abstraction Extension

Thus far we have described continuation abstractions under the assumption that the interfaces to those abstractions would be unchanged from their predecessors' interfaces. In the case of subsystems, we assumed the set of handlers was unchanged. We now investigate the circumstances in which it should be permissible to change the interface, as well as the implementation, of a subsystem during dynamic replacement.

Any changes to the interface can be thought of as some combination of adding new handlers and deleting existing handlers. We can describe situations in which such changes might be desired: it is realized after a module is in use that some additional function is required; development on a system is still underway and the implementors would like to install a service with the option of adding functions as they become ready; requirements change over time and some new functions are desired, while some existing ones are no longer used. Furthermore, while we want

to provide the extended interface to new clients wishing to use it, we also want to ensure that clients using the old interface will be able to continue to use the subsystem through that interface.

We will be concerned here primarily with changes involving only the addition of handlers. Deleting handlers has the obvious problem of existing clients using those handlers. Such a change will be invisible to clients only if no clients use the deleted handlers.

To understand the implications of interface extension, we look at an example. Suppose we want to add a delete_user handler to the mail system. We would argue that such an extension is useful. In fact, it is one that would be likely to be made dynamically, since its omission would not cause immediate problems upon installation of the subsystem, and so could be forgotten, or just left unimplemented at first.

Will clients that do not use the new handler be able to detect the replacement? In this example of extension, there are two ways the extension might be visible to clients. The first problem is that the mailer contains an add_mailer handler that returns an object of the mailer interface type. If a mailer is replaced with an instance of an extended_mailer type containing a delete_user handler, and the add_mailer handler in that subsystem definition returns an extended_mailer object, then clients connected to the original mailer will no longer be able to use the add_mailer handler because the object returned is not of the type expected. This problem will arise whenever a handler (but not a creator) of a guardian returns an object of that guardian's interface type. The problem occurs not only with subsystem extension but with any replacement using a different replacing abstraction. The formal model to be presented in the next section will make clear exactly when this limitation arises. A more precise definition of the cases in which

this problem arises, and a discussion of possible solutions, will be presented in section 4.3.

The second way in which an extension can become visible is as a side-effect of using the new handler. If no client uses the new handler, the results of invoking read_mail, send_mail or add_user will be indistinguishable from those prior to replacement. However, as soon as any client makes use of the delete_user handler, the extension will become visible to other clients. For example, clients might have written programs that depended on the fact that if a user existed at one point, it existed forever. Thus, a program that checked for existence of a user initially, never had to check again, and the program could have been proved correct. Once a delete_user handler is added, this assertion upon which client programs depended is no longer valid.

Whether an extension will be visible to existing clients will depend on the kinds of properties that can be specified about an abstraction, and the kinds of assertions that clients can make based on those specifications. We will not be defining specification or assertion languages in this dissertation. Instead, we will informally describe several classes of extensions, and give intuitive explanations of when such extensions will meet certain safety criteria, so that users wishing to extend a subsystem will have criteria upon which to evaluate the safety of such an action.

The first class of extensions is the addition of query operations. Since queries are read-only, they have no effect on the subsystem's state. An example of such an extension would be the addition of a handler *list_mail(username)* to the mail system. Clients with access to only the original, restricted set of operations should not see any change from the installation of this kind of extension. However, it is true that clients will now be able to see more information than they previously could. If the assertion language associated with the original specification allowed a client to

deduce that the additional information was invisible, this kind of extension would not be safe either. For example, if it were possible to assert that no one could determine from whom a user received mail unless they read that mail, then the addition of a list_mail handler would violate assertions on which existing clients depend. Another example of an assertion that could be violated by addition of query operations is an assertion that a certain event occurs only directly following another event. The introduction of a query operation might change that relationship.

The second class of extensions is the addition of operations that allow some update operations but do not affect any values visible through the original set of operations. The crucial property of this class is that, though the new operations update state, the extension remains invisible to clients that do not use the extended operations. Such extensions usually take the form of adding components to the concrete state and adding handlers that operate only on those components. An example of such an extension to the mail system would be the addition of handlers set_last_use and last_use, which record the date the user last accessed the mailbox and return that date, respectively. While these handlers affect the state of the subsystem, they are independent of the other handlers. Again, if it were possible to write assertions about properties other than the values returned from various invocations, we could not guarantee invisibility of such an extension. The above two classes fall into a category we will refer to later as *non-interfering extensions.*

The third class of extensions is the one for which it may be difficult to guarantee that the assertions on which existing clients depend will not be violated, although this class includes many extensions that would be useful. This category involves adding handlers that interact with the original handlers via the data in the subsystem. Adding a delete_user handler to the mailer is an example. The significant property of this class is that if any client uses the new handlers, the extension may be visible to clients that use only the original set of handlers.

In describing our formalism later, we will define correct replacement for history-independent extensions only We will not forbid either type of extension in the replacement system described later, but rather, leave it to the implementor to determine whether an extension is safe.

4.2 Formalizing Correctness Conditions for Replacement

In the previous section we described the situations and relationships that must be modeled in defining criteria for dynamic replacement. We now present more rigorous definitions of the legality of replacing one subsystem instance with another. We first present the model of abstractions and implementations that we will be using, and discuss what it means for an implementation to *satisfy* an abstraction.[5] We then present the criteria for correctness of *continuation abstractions*, and finally, for *continuation instances.*

4.2.1 Basic Definitions

Associated with any abstraction is a set of events that can occur at the abstraction's interface. There are different ways of characterizing those events. We will define the event set of an abstraction as a set of instantaneous events divided into input and output events. For subsystems in Argus, the set of events that "cross the interface" of the subsystem are the handler invocation messages, the replies to those invocations, transaction commit and abort messages, and messages between the subsystem's background code and the subsystems used by that code. As an example, Figure 4-1 shows the event set for the mail subsystem.

---

[5]The model described here was originally derived from a model developed by Stark [15].

Input Events:[6]

create
∀user∈userid, msg∈message ,{**send_mail**(user,msg)}
∀user∈userid{**read_mail**(user)}
∀user∈userid{**add_user**(user)}
∀node∈nodeid{**add_mailer**(node)}
Commit_Action
Abort_Action

Output Events:

∀mail-instance∈mailer{**createreply**(mail-instance)}
∀am∈ atomic_array[message]{**read_mail_reply**(am)}
read_mail_signal(no_such_user)
send_mail_reply()
send_mail_signal(no_such_user)
add_user_reply()
add_user_signal(user_exists)
add_mailer_reply
commit_acknowledge
abort_acknowledge

Figure 4-1: Mail System Interface Events

Note that a handler invocation and the reply to that invocation are two separate events; the handler executions themselves are internal to the subsystem. Concurrency within the subsystem will be visible as nondeterminism in the order in which the events occur. Thus, input and output events are not obviously paired; several invocation messages could for example enter the subsystem before any replies occur, and replies to invocations need not occur in the order of the corresponding invocation messages. For our purposes here, it will not be necessary to discuss the relationship between input events and output events. [15]

---

[6]Every event has associated with it a transaction identifier, though we have left that identifier implicit in the descriptions here.

An abstraction can be represented as a set of abstract states, a set of events, and a mapping from the states to the sequences of future events that can occur starting from that state:

abstraction = <AS, E, Futures>

where:
AS = {AbstState}
E = {events}
Futures: AbstState --> {eventseq }

and AbstState, eventseq are sequences over E.

An abstract state is, intuitively, the history of events that have occurred in the past. Futures defines the allowable sequences that can follow that history. Note that our definition of abstraction incorporates state, for the reasons discussed in the previous section. Hence, there is no longer an assumed initial state reflecting a null history. AS defines the possible starting states of an instance of the abstraction.

Similarly, we define an implementation to be a 4-tuple:

<A, CS, AF, Impl>

where:
A is an abstraction
CS = {ConcState}
AF: ConcState --> {AbstState}
CFutures: ConcState --> {eventseq}

AF is the abstraction function that maps the concrete states of the implementation to the abstract states of the abstraction. The abstraction function maps a concrete state to a *set* of abstract states because the implementation may lose some information about the actual sequence of events that has occurred, and it may be impossible to distinguish among several abstract states given the information in the

concrete state. Two implementations of the same abstraction will have different abstraction functions. A concrete state defines an equivalence class of abstract states under possible futures. For simplicity, we do not distinguish between the abstract and concrete event sets.[7]

CFutures maps from a concrete state to the set of event sequences that could occur in the future if the implementation were started from that concrete state.

We now need a definition of when an implementation *satisfies* an abstraction. There are numerous possible interpretations for implementation correctness. We do not select a single definition here; rather, we discuss some of the possible interpretations, so that we can determine how the criteria for replacement are affected by each, and how those criteria must be defined in systems using different definitions of implementation correctness.

We first define an implementation to be a *consistent implementation* of its abstraction as follows:

ConsImpl: implementation --> boolean

ConsImpl(I) =
$\forall c \in CS_I [\forall a \in AF_I(c) [CFutures_I(c) \subseteq Futures_A(a)]]$

Intuitively, an implementation is consistent with its abstraction if the following condition holds: the event sequences that can be generated by the implementation from any given concrete state, must be a subset of the event sequences allowed by the abstraction from *each* of the abstract states associated with that concrete state.

---

[7]Stark [15] discusses the relationship between the event sets of the component guardians of a subsystem and the event set of the subsystem.

This definition states only that any action performed by the implementation is allowed by the abstraction. An implementation that did nothing would satisfy this predicate.

This criterion could be strengthened in several ways. There are two we have considered. One is imposing requirements on the sequences of input events accepted by the implementation. The other is imposing requirements on the set of starting states of the implementation.

Requirements on the inputs accepted are intended to ensure that the implementation performs the functions defined by the abstraction, in addition to the consistency constraint that states that if it performs a function, it does so correctly. We therefore insist that all inputs accepted by the abstraction are accepted by the implementation. One way to ensure this property is suggested by Stark [15]. In his formulation, both abstractions and implementations are constrained to accept any input event at any point. Hence, the following properties hold on futures:

Admissible_Abst: Abstraction --> Boolean

$$\text{Admissible\_Abst}(A) = \forall as \in AS_A \; \forall e \in E_A^{input} [\, \forall p \in \text{prefix}(\text{Futures}_A(as)) \; [\, p \bullet e \in \text{prefix}(\text{Futures}_A(as))]]$$

where prefix({eventseq}) denotes all event sequences that are prefixes of a sequence in {eventseq}.

Similarly, we define:

Admissible_Impl: Implementation --> Boolean

$$\text{Admissible\_Impl}(I) = \forall cs \in CS_I \; \forall e \in E_A^{input} [\forall p \in \text{prefix}(\text{CFutures}_I(cs)) \; [p \bullet e \in \text{prefix}(\text{CFutures}_I(cs))]]$$

The motivation behind this constraint is that modules have no control over the senders of input events, and hence the input events will occur even if the module chooses not to respond to them. Thus, constraints are imposed, not only on implementations, but on abstractions as well. Since the implementation must now accept all inputs, and by ConsImpl must respond correctly to those inputs, implementations that "do nothing" will no longer be considered to satisfy the abstraction.

An alternative formulation leaves it to the abstraction definition to determine when input events can occur, and defines implementations to be required to accept input sequences allowed by the specification. This requirement ensures that any inputs allowed by the abstraction are handled by the implementation. This constraint would be expressed as:

InputConsistent: Implementation --> Boolean

InputConsistent(I) =
$\forall cs \in CS_I [\forall as \in AF_I(cs) [CFutures_I(cs)|E_{A_I}^{inputs} = Futures_{A_I}(as)|E_{A_I}^{inputs}]]$

where {seq}|{event} is the set of all sequences obtained by taking the sequences in {seq} and removing all events not in {event}.

The other possible criteria we consider for an implementation satisfying an abstraction are requirements on the relationship between the set of concrete states in the implementation and the abstract states in the abstraction. Our definition of abstraction, by allowing many possible starting states, complicates the analysis of this relationship. As stated earlier, without replacement, there is an initial state for an abstraction, and the implementation must start in that state. We now define a set of abstract states, and the future behavior possible from each of those states. The question then arises as to whether an implementation must be able to start in any of

those states, i.e. must the abstraction function map *onto* the abstract states. More precisely:

FullImpl:Implementation --> boolean

$$\text{FullImpl}(I) = \forall\, as \in AS_{A_I}\, [\, \exists\, cs \in CS_I\, [\, as \in AF_I(cs)\,]\,]$$

This definition would exclude many implementations that we intuitively consider correct implementations of an abstraction. In particular, it limits the choices in implementing non-deterministic specifications. In our unique-id generators, the implementations we were discussing earlier might not satisfy FullImpl, because each could generate some subset of the possible sequences that satisfied the uniqueness criterion, but would not generate all such sequences. Therefore, there are abstract states that certain implementations might never reach. (Recall that an abstract state is simply the sequence of past events.)

We have presented possible criteria for input acceptance, and one possibility for defining state completeness. We will not choose among these options here. In the remainder of this chapter we will use ConsImpl, the least restrictive, to indicate that an implementation satisfies an abstraction.

4.2.2 Definition of Correct Replacement

In this section we describe the conditions on an abstraction that will allow an instance of that abstraction to replace a current instance of a possibly different abstraction. We then define the relationship that must hold between the replaced instance and the replacing instance, assuming that the conditions on the replacing abstraction hold. We first examine replacement without allowing extensions to the abstraction, and then modify our correctness conditions to allow extension.

The intuitive condition for correctness is that the new instance generate only futures

that would have been permitted by the replaced abstraction, as continuations from the state in which replacement occurred. However, as described in section 4.1, at the time of replacement, we will know only a set of possible abstract states, not a single state, since we no longer know the exact history that has occurred. We therefore first characterize the set of futures allowable from a set of abstract states:

ReplFutures: Abstraction, {AbstState} --> {eventseq}

$$\text{ReplFutures}(A, \text{States}) = \bigcap_{a \in \text{States}} \text{Futures}_A(a)$$

The set of allowable futures after replacement is the intersection of the futures allowed by any of the possible states. Since we do not know the exact history of the module, we can only permit those futures that can legally follow from any of the states.

We can now define the conditions that must be satisfied for an abstraction A2 to replace an abstraction A1, when A1 is known to be in one of a set of abstract states.

AcceptableRepl: abst, {AbstState}, abst --> boolean

AcceptableRepl(A1, States, A2) =

(1) $\exists\, s \in AS_{A2}$ [$\text{Futures}_{A2}(s) \subseteq \text{ReplFutures}(A1, \text{States})$] and

(2) $\text{InputsOf}(\text{Futures}_{A2}(s)) = \text{InputsOf}(\text{ReplFutures}(A1, \text{States}))$

where:

InputsOf: {eventseq} --> {eventseq}
$\text{InputsOf}(ES) = ES \mid \text{Events}_{ES}^{\text{inputs}}$

and:

$\text{Events}_{ES}^{\text{inputs}}$ is the set of input events that occur in the sequences in ES.

The first condition states that there must be some state of the replacing abstraction that allows only future sequences that would have been allowed by the replaced abstraction, had it continued. The important point here is that the starting state of the replacing abstraction does not have to represent the same history of events that was represented by the ending state(s) of the replaced abstraction. Intuitively, the starting state of the new abstraction and ending state of the old abstraction must be "equivalent" with respect to possible futures, but not necessarily with respect to possible past sequences.

The first condition thus ensures partial correctness: any action allowed by the replacing abstraction would have been permitted by the replaced abstraction. It does not, however, guarantee that the replacing abstraction will permit the same range of functions. This condition alone is therefore insufficient to guarantee that replacement will be invisible to clients. The second condition ensures that the replacing abstraction permits the same set of invocations as its predecessor. It states that all sequences of input events permitted by the replaced abstraction must be permitted by the replacing abstraction. It does not require that all of the "replacement futures" (as defined by ReplFutures) be retained; however, the the two conditions together imply that, for every input sequence, the replacing abstraction must retain at least one sequence containing that input sequence that was permitted by the replaced abstraction.

Thus far, we have ignored replacement with extended abstractions. We now modify the above definitions to allow *non-interfering extensions.*

AcceptableExt: Abstraction, {AbstStates}, Abstraction

AcceptableExt(A1, States, A2) =

(1) $\exists\, s \in AS_{A2}\, [\, Futures_{A2}(s)|E_{A1} \subseteq ReplFutures(A1, States)]$ and

(2) $InputsOf(Futures_{A2}(s))|E_{A1} = InputsOf(ReplFutures(A1, States))$

This definition is identical to the definition of acceptable_repl, with the exception that, when comparing future sequences allowed by the replacing abstraction with those allowed by the replaced abstraction, we ignore the existence of the new events. When the future sequences of the replacing abstraction are restricted to the event set before extension, they must satisfy the conditions for AcceptableRepl.

We have now provided the definitions for legal abstraction replacement. The next step is to determine the conditions under which one subsystem *instance* can replace another. We define an *instance* to be a pair <implementation, ConcState>. The pair <I, cs> can be mapped to the set of event sequences denoted by $CFutures_I(cs)$.

For one instance to correctly replace another, the state in which the replacing instance starts must "correspond" to the state in which the replaced instance stopped.

LegalStartState: instance, {eventseq} --> boolean

assume i2 = <I2, cs2>

LegalStartState(i2, ES) =
$\exists$ a2 $\in$ AF_{I2}(cs2) [$Futures_{A_{I2}}$(a2)|$Events_{ES}$ $\subseteq$ ES]

This definition states that the starting state of an instance is a legal starting state with respect to a set of event sequences, ES, if there is some abstract state associated with the starting state, from which all allowable future sequences are in ES. If the implementation associated with the instance is *consistent* (as defined by ConsImpl), and the starting state of the instance is a legal starting state, it follows that every event sequence that can be generated in the instance is contained in ES: ConsImpl states that, from any concrete state, the implementation can allow only future sequences in the intersection of the sequences allowed by the associated abstract

states. Hence if the *Futures* of any one of the abstract states is contained in ES, the sequences generated by the implementation must be in ES.

We now define the correctness of replacing one instance with another as follows:

InstanceRepl: Instance, ConcState, Instance --> boolean

Assume:
i1 = <I1, cs1>, I1 = <A1, CS1, AF1, CFuture1>, A1 = < AS1, E1, Future1>
i2 = <I2, cs2>, I2 = <A2, CS2, AF2, CFuture2>, A2 = < AS2, E2, Future2>

InstanceRepl(i1, cs, i2) =

(1) ConsImpl(I2) and
(2) AcceptableExt(A1, AF1(cs), A2) and
(3) LegalStartState(i2, ReplFutures(A1, AF1(cs)))

Intuitively, an instance of abstraction A2 can replace an instance of abstraction A1 in concrete state cs, if the new instance is a correct implementation of A2, A2 is a correct replacement of A1, and finally, the state in which the new instance is started "corresponds" to the state in which the old instance ended (i.e. both concrete states map to the same set of allowable futures).

4.3 Discussion

The formal model presented in this chapter enables us to characterize more precisely those cases described in section 4.1 that do not concur with our initial intuitions about the safety of replacements. In addition, the model brings to light several issues regarding the semantics of replacement that were not previously clear. In the first subsection to follow, we discuss the anomaly, exemplified by the unique-id generator, in which two correct implementations of an abstraction are not *replacement compatible*, i.e. one cannot be used to replace the other dynamically. In

the second section we describe a problem that becomes apparent in examining our definitions for acceptable replacements: successive replacements in a module may restrict the allowable future behaviors further than the restrictions imposed by past events alone. Finally, we discuss the use of abstraction replacement in Argus, and some limitations of abstraction replacement not immediately obvious from our earlier discussion.

4.3.1 Incompatible Implementations

The examples of the unique-id generators presented earlier illustrated that two correct implementations of the same abstraction might not be "replacement compatible", i.e. it might not be possible to replace an instance of one with an instance of the other. With the formal model presented earlier, we can characterize the properties of subsystems that give rise to these counter-intuitive situations.

The criteria for instance replacement (InstanceRepl) indicate when one implementation will be able to replace another implementation of the same abstraction. There were three conditions in InstanceRepl: the implementation had to satisfy its abstraction, the abstraction had to be a valid replacement abstraction, and the state in which the implementation was started (defined by the instance of the implementation) had to concur with the ending state of the previous instance.

Since we are assuming here that the abstraction is unchanged (and an abstraction is always a valid replacement of itself), and that the implementation is correct, then if one implementation cannot replace another, there must be no legal starting state for the replacing implementation. There are two ways in which this situation can arise. First, it is possible for one implementation of an abstraction to end in a state that another implementation could never reach; there may then be no state in the second implementation's set of concrete states that maps to only legal future sequences.

The second situation arises when so much history information has been lost from the concrete state that we cannot derive a legal starting state for a different implementation.

Whether one implementation can reach a state not reachable by another implementation, and under what circumstances that situation will arise, are dependent upon the definition of an implementation *satisfying* an abstraction. If we use the definition FullImpl, which insists that every abstract state in the abstraction is reachable in any implementation of the abstraction, this state incompatibility cannot happen. The state in which the replaced instance ended will correspond to some abstract state *as*, and the new implementation must have some concrete state *cs* that maps to *as*. By ConsImpl, the futures from cs will satisfy the abstraction.

If by contrast, we insist only on ConsImpl, then obviously there can be states that the new implementation will not reach, since that implementation need not perform any of the abstraction's allowable behavior. If in addition to ConsImpl, we require InputConsistent, the analysis becomes more interesting. Now the implementation must accept all input sequences allowed by the abstraction from each of the implementation's possible starting states. Hence there is some legal behavior it will perform given any sequence of inputs. Now if we examine how one implementation can arrive at a state unreachable from another implementation, we see that this situation will only arise when there are two legal futures from a given state that have the same input sequence, i.e. when the abstraction is nondeterministic. In Argus, where concurrency control is handled automatically by the system, nondeterminism due to concurrency will not cause this situation: all implementations will be able to reach all states possible from reorderings and concurrent execution of handlers. One implementation will be able to reach a state unreachable from another only when the abstraction allows different possible replies to a given invocation occurring in a given abstract state. In our unique-id generator, the specification we gave

allowed any number of possible id's to be generated next, given any past sequence. Thus, the nondeterminism allows one implementation to generate sequences of id's that would not be generated by another implementation. Since the second implementation would not have generated that sequence, there may be no point at which it could start running and ensure that no duplicate id's would be generated.

The other property that may lead to implementation incompatibility is the mapping of a concrete state to a set of abstract states, rather than a single abstract state. The larger the set of abstract states, the smaller the set of legal futures may become, since ReplFutures is dependent on the intersection of the futures from each of the states. It may be possible that two implementations could both reach the same state (i.e. same past history), but that there is no concrete state in the second that will generate only futures legal for *all* of the abstract states associated with the final state of the first instance. We discuss this situation further in the following subsection.

4.3.2 Behavior Restriction from Successive Replacements

Another aspect of replacement that our formal model enables us to evaluate is the limitation on module behavior caused by successive replacements. There are two ways in which successive replacements cause additional restrictions on behavior: one is the subsetting allowed in replacing abstractions; the other is that the abstraction function may map to wider sets of abstract states because it must take into account all possible starting states of the instance, and all possible ways to have reached each starting state.

The subsetting in replacement abstractions occurs because AcceptableRepl allows the replacing abstraction to permit subsets of the legal futures (ReplFutures). Thus, some legal continuation sequences will be eliminated, not only from this replacing instance, but from all future replacements as well. Obviously, successive

replacements can eliminate increasing numbers of futures allowable according to the module specification. Note that attempting to eliminate this problem by requiring replacing abstractions to permit all sequences in ReplFutures will reduce the set of acceptable replacements.

The second way in which replacement narrows the set of possible futures is due to history information lost from the concrete state. There are two kinds of lost information: information about the actual event sequence that occurred in the instance, and information about prior instances in the module and the concrete state in which this instance started.

The abstraction function of an implementation maps to a set of possible abstract states rather than a single state. The legal futures are then limited to those sequences allowed from any of the states in that set. The larger the set to which the abstraction function maps a concrete state, the more restricted the future behavior. Successive replacements compound this problem in the following way. If several replacements have already occurred, and the starting state of the instance is not known at the time of replacement, the set of histories that could have led to a given concrete state becomes much larger. The reason for the increase is that, if the starting state is unknown, the abstraction function must take into account ways of reaching the current concrete state from any possible starting state of the implementation. Thus, the set of histories may be much larger than actually could have occurred in this instance. By retaining information about the replacement history of a module, we can reduce the set of states to which any concrete state is mapped and thereby increase the set of possible replacing instances.

The unique-id generator exemplifies this situation. Suppose we have two implementations of the unique-id generator described earlier, one of which generates numbers in increasing sequence, and the other in decreasing sequence. If

the current implementation is generating ids in increasing order, an implementation using decreasing order can replace it by starting at a uid that is larger than the last uid by at least X, *if we know that no prior replacements have occurred.* If we have no information about prior replacements, it is possible that the replaced instance itself replaced an earlier instance, which may have been generating larger id's. Hence, we can perform no replacements, because too much history information has been lost. Maintaining a *replacement history* about the past instances in a module, along with their starting and ending states would be necessary to alleviate this problem.

Finally, the set of futures may be narrowed because different implementations generate different subsets of the abstraction's allowable futures. The set of futures the new implementation can generate from the state in which the old implementation stopped might be smaller than the set possible if the same instance continues to run.

For example, suppose from a given abstract state, the sequences abbb, cbbb, and cddd are allowed. The replaced implementation generates only abbb and cbbb, the replacing implementation only cddd. If the sequence ab has been generated prior to replacement, the replacing implementation can produce no legal futures. Hence, there is no instance of the replacing implementation that will satisfy

InstanceRepl(replaced_instance, ab, replacing_instance)

where ab is the concrete state of the replaced instance that corresponds to the sequence ab having occurred.

It is unclear how often these situations will actually result in more restricted future behavior. In our mail system, for example, it is true that not knowing the starting state of an instance will admit many more event sequences leading to a given

concrete state. However, no information critical to determining possible future sequences has been lost from that concrete state. Consequently, the possible futures from any of those event sequences is the same: the intersection will not narrow the set of legal futures. Almost all of the problems are due to nondeterminism in the abstraction. Theoretically, however, it is possible that successive replacements will limit behavior substantially and may be undesirable for that reason.

4.3.3 Replacement Abstractions in Argus

In the previous section, we discussed the limitations of replacing one implementation of an abstraction with another. Here, we examine the problems of replacing one abstraction with another in Argus. The conditions for abstraction replacement are straightforward: a second abstraction is acceptable if it allows only futures that would have been allowed by the replaced abstraction, from the abstract state in which the second is to be started. However, in analyzing how this criterion meshes with Argus abstractions, we find there are considerable practical restrictions on when abstraction substitution is possible. Our model allows more complex relationships among abstractions and implementations (and implicitly specifications) than the Argus type system and library structure permit.

An abstraction in Argus is represented by a type, and objects of one type can never be assigned to variables of another type. Recall from Chapter 2 that each guardian type has associated with it an automatically generated guardian interface type; the guardian objects given to clients are objects of this interface type. If an abstraction is replaced by a different abstraction, then new guardian interface objects generated will be of a different type than previously distributed interface objects. (In practice, this situation is unlikely to arise; a change in abstraction will probably occur only when the interface is to be extended.)

This discrepancy has the following effects. Clients already holding the interface objects of the replaced abstraction can be accommodated because replacement does not involve reassigning those objects to objects of the new type. (The individual handlers can be automatically reassigned since they do not change type.) The problem arises when one of those clients tries to obtain a new interface object from the subsystem it believes is an instance of the replaced type. In the mail system, for example, the add_mailer handler returns a mailer object. If the type changes during replacement, the add_mailer handler will return an interface object of the new type. The client is expecting an instance of the replaced interface type, and an object of the replacing interface type cannot be accepted as the return value, according to Argus type rules. (Although, since type checking is performed only at compile time, it is unclear what the effect of such an invocation will be.)

This situation arises whenever a client expects to receive an object of a replaced subsystem type. It is evident in our formal model in that acceptable replacement abstractions must only allow event sequences allowed by the replaced abstraction. In the case of our mailer, the add_mailer_reply event will change, since the new mailer type will return a different type of object. Thus, no sequence containing that event can be permitted by a replacing abstraction. Replacements of the mail system will therefore be limited to implementation changes (or abstraction changes in which the new type still creates mailer guardians of the original type). This restriction will hold whenever a handler (not a creator) of a guardian returns an object of the guardian's interface type.

Invoking creators is a slightly different problem. Under what conditions can clients that are compiled knowing of one guardian type invoke creators of a replacing type? Again, Argus typing problems arise; the client must know the type to invoke the creator and it would require support in the language to allow substitution of the replacing abstraction's creator automatically. (In addition, the replacing abstraction must be an acceptable replacement from the initial state.)

Replacements with extending abstractions encounter still other problems. We referred to this issue in Section 4.1. Here, not only is the interface type different in name, but it contains a different set of handlers as well. To allow any handler to be added to the mail system requires being able to bind interface objects of the extended type to variables of the original type. This raises numerous language issues, such as whether binding in the other direction would then be possible. The meaning of types in Argus would have to be extended to accommodate the notion of "type compatibility". In this dissertation, we disallow extensions that change the original event set in addition to adding events.

Overall, replacement of one abstraction with another in Argus will be limited to the case in which the interface type does not appear in the interface of any of the guardian's handlers. Furthermore, new clients will have to know the new abstraction type, and old and new clients will not be able to communicate values of the interface types to one another.

Chapter Five

User Requirements

Thus far, we have discussed the requirements imposed on dynamic replacement by the need for type safety and correctness. In this chapter we focus more directly on the mechanism needed to perform replacement in the Argus system. We examine in detail the actions that must be performed to effect a replacement, and the kind of support users require in the course of performing a replacement. We distinguish between those actions that can be performed automatically and those that require user intervention.

Because subsystem instances are not identifiable in the running system, the actual replacement mechanism will have to manipulate guardians individually. This limitation is reflected here in the descriptions of the actions to be performed; the functions described are for single guardians rather than subsystems.

We first address the problem of identifying the guardian instances to be replaced and locating those instances in the Argus network. This problem is the first one encountered by a user attempting to perform a replacement. While it is not a problem unique to replacement(for example, it arises in debugging also), it must be handled if replacement is to be accomplished. Because the problem is only tangentially related to replacement and much overhead is involved in supporting a full solution to the problem, we provide only a minimal solution here.

We then examine each aspect of a single guardian subsystem that may be changed during replacement, and the actions needed to accomplish that kind of change. These include changes to program text, state structure, and finally location. We

then go on to discuss issues that arise in replacing multi-guardian subsystems as single entities.

This chapter deals only with enumerating the requirements of the replacement system's users. In chapter 6, we will propose a mechanism for supporting these requirements.

5.1 Locating Subsystem Instances

Identifying the subsystem instances to be replaced and locating their components is a major difficulty facing the user at the start of replacement. This section identifies the properties of subsystem instances that distinguish one instance from another and might influence whether or not a given instance should be replaced. From this information, it is possible to determine the set of functions that must be provided to allow the user to locate the instances needed.

One of the characteristics of Argus that complicates this problem is that neither guardians nor subsystems have any user-meaningful names. These modules are created by other subsystems dynamically, and are distinguished from one another within the Argus system only by the fact that different clients hold the handler names for different instances, and that the states of various instances are different at any given time.

This situation presents two problems. First, given that several subsystem instances exist, how does the user determine which it is that must be replaced? The first function of the replacement server therefore must be to give users a means of distinguishing between instances of subsystems in existence. The second problem is how the user can locate the components of a given subsystem instance if the system cannot identify that set of components as a unit.

5.1.1 Identifying Subsystems using System Information

A user entering the replacer must first be given some method for finding out what subsystem instances exist. In providing the subsystem location facilities for the replacement server, we must know what information users will need to be able to make this decision, and incorporate in the replacement mechanism the means to obtain that data. Certainly the first piece of information users need in differentiating between subsystem instances is the type of those subsystems: the user is interested in replacing a subsystem instance of a given type with another instance of that subsystem. Thus, the replacement system will need to provide some method of locating instances of a given type; the catalog provides one such method.

In addition to the type of the subsystem, users will generally be interested in finding instances using a particular implementation. For example, if the replacement is taking place to fix a bug, then the instances in which the user is interested are those using the incorrect implementation.

Aside from the type information for a subsystem or guardian instance, the other obvious piece of information that is readily provided by the system is location of a guardian. (Since subsystems may span nodes, the location is meaningful only with respect to individual guardians.) Whether a subsystem needs to be replaced may depend on the node(s) at which its components reside. For example, we might want to relocate a maildrop if it is "too far away" from most of its users, given the current network topology.

Thus, the type, the implementation (including bindings to externally defined types), and the location are the pieces of data known to the system that are most likely to be used in identifying a subsystem or guardian instance for replacement.

5.1.2 Identifying Subsystems by State Information

Another requirement arises in attempting to locate subsystems for replacement. Whether a subsystem should be replaced may depend on the value of its state. Removing all instances of that type to examine their state may cause significant disruption to the system. It would therefore be desirable to provide a way to examine such state without stopping the guardian in question. Suppose for example, in our mail system, we wanted to replace maildrops having more than a given number of users with new versions allowing faster access. It may have been decided that it was not worth replacing maildrops with fewer users since the benefits would not be as significant.

The ability to examine state in this manner serves two other purposes as well. First, it will aid in locating subsystem components, since those components may be static components of the root's state (or some other component's state). The second case for which examining state is useful is locating components of subsystems with cluster roots. In this case, there is no root guardian to search for, and finding all the components of a single subsystem instance may require accessing a client's state. To avoid disrupting clients for replacement, users should be able to access the state while the client guardian is running.

Information obtained while a guardian is running cannot be time dependent in any way that will affect the decision on whether to replace the guardian. Since the guardian is still running, it can be participating in transactions and the state value may change between the time the state is examined and the time the replacement is actually started. The user must know that needed state information is static or that the effect of the interceding transactions will not be significant in the decisions made based on the data. If the state value were time dependent, exclusive access to the guardian's state would be needed until replacement started or a decision was made not to replace.

In our maildrop replacement, for example, while the number of users may change over time, once it has reached the specified number, it will not be reduced. Hence, there is no semantic reason to stop use of all maildrops while their states are examined. On the other hand, stopping user access to all maildrops will disrupt service.

In summary, to locate subsystem instances in the running system, we require that information be available as to the guardian instances in existence at each node, along with the identity of their types and implementations. Furthermore, while not absolutely necessary, it is helpful to be able to examine the permanent state of a guardian without disrupting its availability.

5.2 Single Guardian Replacement

This section examines the tasks users will have to perform to replace a single-guardian subsystem. To identify those tasks, we analyze the various changes likely to be made to module definitions in the library, and the corresponding modifications that must be made to active instances of those modules to incorporate such changes dynamically. The set of changes that can be made is limited by the legality constraints in Chapter 4. By enumerating these classes of changes, we can derive the set of functions that must be provided to users of the replacement system. In the first two subsections we describe the functions required to modify guardians at the user's site. In the final subsection, we discuss relocation of guardians and the management of replacements at remote sites.

5.2.1 Code Modification

One of the simplest kinds of changes that requires dynamic replacement is a modification to the code of a single guardian with permanent state. If the code change does not affect the representation or value of the permanent state objects, that state need not be modified. Before replacement starts, the user must install the new implementation and external binding information in the library (see section 2.5). The replacement mechanism is used to convert running instances to the new implementation. Such a conversion requires the following to occur: the new code must be loaded; the existing stable state must be associated with the new instance; and a mapping must be defined from handlers of the replaced guardian to handlers of the new guardian, so that messages can be automatically redirected to the new guardian. The old instance must of course be removed from service. The replacer should be checking that the new handlers and state are of correct type before allowing the replacement to proceed. Finally the new guardian must be made available to clients. Since we are only transferring the stable state to the new guardian, we are making the assumption that the new guardian is " returned to service" as if the guardian were recovering from a crash the recovery code must be invoked to reinitialize the volatile state.

An illustration of this kind of replacement can be seen using the mail subsystem defined in Figure 2-5. Suppose for instance that the user wished to change the procedure used to select a maildrop for a new user. (This is the select handler in the registry.) Note that only a registry guardian, not the entire mail system, must be replaced to make this modification.

No modifications are being made to the state of the registry. However, the replacement mechanism must ensure that the state value is correctly transferred to the new instantiation. The state that must be moved to the new instance is only that portion declared to be *stable*, i.e. *steers* and *regs*.

This case illustrates why we cannot use the replacement method of starting the new version in parallel with the old one, and transferring clients over gradually. The new version must start with the current *regs* and *steers* values from the old instance. If the old instance continues to run after the new one starts, the two instances will be making independent changes to their separate copies of the state. Thus, one will be recording the results of some transactions while the other will be recording results from other transactions. When the old instance is finally removed, the history it recorded that had not been transferred to the new instance will be lost. Hence, the switch from the old to new instance must be performed atomically with respect to clients of the guardian.

The replacement mechanism must ensure that all handler names for the old instance must be associated with the new instance, so that clients holding those handler names need not be aware of the replacement. Thus, for example, the all_regs handler from the old instance must be bound to the all_regs handler from the new instance. In general, clients should be unaware of the replacement; the only visible effect should be the unavailability of the mailer for some period of time. Clients can continue to send messages to the handlers they have invoked in the past; the rerouting should be handled automatically.

Thus, in summary, we can identify several requirements imposed on a replacement mechanism by the need to make changes in the program text of modules on-the-fly. These are:

- The modification must be atomic with respect to clients.
- The stable state must be transferred to the new instance.
- If state transfer is to be limited to stable state, it must be possible to invoke the recovery code when guardians are restarted after replacement.

- The handler connections must be transferred to the new instance.

5.2.2 Modifying State Structure

The problem becomes more complex when the changes to the implementation involve changing the form of the permanent state in the guardian. For example, consider the maildrop guardian in the mailer. Suppose the user decided that accessing the userlist was too slow. One possible improvement would be to alphabetize the list and use a binary instead of linear search technique for locating users. Such a change would involve not only changing the implementation of the maildrop's handlers, but (if we are to preserve the current state information) sorting the list of mailboxes (*boxes*) that already exists. The former change can be performed statically by inserting a new implementation in the library. The latter change must be performed by the replacement service. Notice that this change is entirely a change in implementation. The interface to the maildrop remains the same. To be able to perform this replacement, the implementor must be able to access the state (i.e. boxes) at the time of replacement and change its form (to the alphabetized list). This change should be invisible to clients of the guardian since the abstract value of the state is unchanged. It will be legal according to the criteria from Chapter 4, since the result of any input event, and the set of allowable future sequences, are unchanged by alphabetizing the list.

It is clear from the above example, however, that the change in the state is application dependent. The replacer could not automatically transform the state; the user must explicitly sort the old list. Thus, one requirement for the replacer is that the user be able to invoke arbitrary procedures on the state. This requirement raises reliability issues. The user is modifying state on which consistency constraints exist and for which the transaction system has guaranteed robustness. An incorrect transformation is possible, and the result of such an error will be the loss of state

consistency. Hence, we cannot ensure that our instance replacement criteria (Section 4.2) are satisfied without verifying the translation procedure. The only reliability checking we can perform is type checking; we can guarantee that the objects stored in the new state are of the types expected and that any procedures used to modify the state are type-correct.

Another point evident from this last example is that a replacement needed for one instance of a guardian definition might not be useful for all instances. Different maildrops in the mailer might have different usage characteristics, for instance some having more users or heavier load characteristics. It might be important to have different implementations for different instances. For example, it might only be worth changing to sorted lists for those maildrops that have many users and frequent access. Otherwise, the overhead of sorting becomes unnecessary.

Hence, we add another functional requirements for the replacement mechanism:

- The ability to invoke user-defined transformation procedures on state objects.

5.2.3 Internode Replacement

There are two aspects to internode replacement of single guardian subsystems: relocating the guardian to a different node in the network, and controlling replacement of guardians at sites other than the one at which the user is located.

If a guardian must be moved to a different node, but its state must be retained, the move can be accomplished by replacing the guardian with another guardian of the same type at the new location. Such a replacement would be necessary if, for instance, a site were about to be removed from the network for an undetermined period of time and there were services at that site that had to be kept available.

Alternatively, usage patterns may change in a way that makes it useful to have the module closer to others in a different location. For instance, if maildrops didn't exist at every node and one of them was originally placed at a node that had few users, it might speed up mail delivery and reduce network traffic to move the maildrop to the location where most of its users were.

The first problem that arises in supporting this kind of replacement is that the state needs to be moved to the new location, but the objects that comprise the state may not be *transmissible* (see section 2.3). If the values in the state never need to be transmitted during the normal operation of the guardian, there would have been no need to use types with encode and decode operations for the state objects. Hence, it will be necessary to provide some mechanism whereby the user performing replacement can transform the state objects into transmissible ones, so that the needed values can be moved. At the new site, the transmitted values will have to be transformed into objects of the types used by the new instance.

The second problem arising in guardian relocation is one we refer to as *type incompatibility.* Because of the requirements of node autonomy in the Argus system, there is no assurance that types that exist at one node will have implementations for other nodes in the network. Therefore, relocating a guardian may require changing the implementation, as well as moving the state. There may be types in the original implementation's state that do not exist at the new site, and conversely, the types used in the new state structure may not exist at the old site. Even if the state objects are transmissible, they cannot be sent to the new site if that site does not recognize their types. The objects will have to be translated at their existing site into a form that is both transmissible and recognized at both sites. Again, at the new site, yet another translation may be required to derive the objects for the new state.

Controlling replacement at remote sites, even when the guardians are not being relocated, involves many of the same problems. It may be necessary to translate the state of the replaced guardian into a new format. It is inconvenient to first move that state to the user's site to be manipulated locally because of nontransmissibility or type incompatibility. Therefore, the user must have the ability to specify commands and procedures to be invoked at the guardian's site. This ability differs from remote procedure call in that the results are not to be returned to the user's site; rather they are used at the site of the invocation. Thus, facilities are needed for naming and storing these values at the guardian's site, and then accessing them in later invocations. These requirements arise in relocating guardians between two remote sites as well. In Chapter 6, we will have to define replacement environments that support these functions.

Thus, the following additional mechanism is needed to support internode replacement:

- The user must have the ability to manipulate state objects and invoke procedures at remote sites during replacement.

- It must be possible to store and access temporary values at remote sites during replacement.

5.3 Multiple Guardian Subsystems

The next category of changes involves multiple-guardian subsystems. The library information about subsystems will provide information about the interfaces to those subsystems. As with single guardians, only the interface must remain unchanged across a replacement. All implementation details should be modifiable. Users should be free to change any such details, including such major changes as the types of the component guardians. It should not matter if those guardians are the same

types as those in the earlier version, as long as the handlers in the subsystem interface are provided by the new set of guardian. The handlers used internally in the subsystem need not stay the same. Similarly, although the abstract value of the state must remain unchanged across a replacement, the state information may be redistributed among the components of the subsystem in a completely different configuration than in the old version. Unfortunately, we will not be able to meet this goal entirely: the root guardian type will not be permitted to change. This restriction is due in part to the system of storing the subsystem information in the root guardian, and in part to requirements of the Argus typechecker.

One such change would be to replace the mail system with one having mailers and a new type of guardian called *mailsite*, where mailsites perform all the functions of both registries and maildrops. The structure for the mailsite type, and the new mailer implementation are sketched in Figure 5-1.

```
mailsite = guardian is create
                handles add_mailsite, add_user, read_mail, send_mail, select, all_sites

steering = struct[ site:mailsite, users:userlist]

steeringlist = atomic_array[steering]
site_list = atomic_array[mailsites]

stable steers:  steering_list

add_mailsite = handler(home:node) returns (mailsite)
add_user =     handler(usr: user_id)
read_mail =    handler(usr: user_id) returns (msg_list)
send_mail =    handler(usr: user_id, msg:message) signals(no_such_user)
select =       handler() returns (mailsite)
all_sites =    handler() returns ( site_list)

end mailsite
```

Figure 5-1: A Restructured Mail System

The user interface to the mailer is unchanged; however, the internal handlers of the mailer are different. Add_maildrop and add_registry are replaced by the single handler add_mailsite. In addition, the state of the mailer now points to a mailsite rather than a registry, and the code for all of the mailer's handlers is different, since mailsite handlers must now be invoked instead of registry and maildrop handlers. All of the handlers of the registries and maildrops must be permanently removed. The state information from the maildrops must be moved into the mailsites. It will be the user's responsibility to ensure that the set of mailsites instantiated during replacement start with the correct knowledge of the other mailsites. In addition, the steering_list will have to be reconstructed using the old steering_list and the information as to which maildrop states were moved to which new mailsites. The user must decide explicitly which registry and maildrop instances have state moved to which mailsites. Since these instances are at various sites in the network, some of the state information will have to be moved to the sites of the new mailsites. Performing this replacement would obviously be time consuming, as it involves every component of the mail system, which we assume to be network-wide. Whenever a subsystem structure involves replicated data, or has consistency constraints among large numbers of components, replacements will also frequently involve large numbers of components, since the internal consistency constraints must be maintained.

Performing this kind of replacement requires combinations of the actions required for the various kinds of single guardian replacements. Little new power is required. It now must be possible to rebind some handlers from a guardian to one new guardian instance, and other handlers from the same guardian to another instance. The two new instances might not even be at the same node. Similarly, it might be necessary to move various state components of the same instance to different locations. The same problems of state relocation arise when multi-guardian

replacements, span nodes. as arise in guardian relocation. Other than these generalizations of previous requirements, the only special support needed for multiple guardian subsystems is the library support described in Chapter 3.

5.4 Subsystem Extension

Finally, there will be cases in real systems in which it is discovered after a subsystem is running that some needed function has been omitted. As discussed in section4.1.1, type extension will be permitted. The replacement system, however, will not distinguish between non-interfering extensions and more general extensions. The user performing an extension must be aware of the implications of extension as described in section 4.1.3. The only checking that can be performed is to ensure that the guardian type being replaced does not appear in the handler interface of that guardian type, as discussed in Section <SpecsLimit>. The main functional requirement for extension is that the user be allowed to create an instance of the extended type and rebind existing handlers to some of the handlers of that instance, while allowing the others to be newly created.

5.5 Limitations

There are also cases of replacements that would be considered legal according to our criteria, but which are particularly difficult to accomplish. As stated in Chapter 3, subsystem information is distributed between the clients and the subsystem's guardian components. Information that is replicated in every client cannot be changed without locating each of those clients. Thus, the more information residing in the guardian components of a subsystem, the greater the range of possible replacements. Thus, replacement of cluster-based subsystems that affect data residing at client sites is particularly difficult.

There is one kind of legal replacement that requires locating the clients when the only information they hold is the handler names for the subsystem. This is the situation in which some clients are to be reconnected to one handler and others to another handler: in other words, when a subsystem is being split. If, for example, in the mail subsystem, we decided that a maildrop had become too large, we might want to split it into two maildrops. Half the users currently in that maildrop could remain. However, the other half would now be in the new maildrop. The state in each registry has a maildrop listed for each user. If two maildrops were being combined, the handlers for both could be rebound to the same handlers. The steerlists would not have to be updated in each registry. Only the two maildrops in question would have to be accessed during replacement. In this situation, the registries are in some sense the clients for the maildrop subsystems. The merging of two maildrops could be performed without locating those clients. However, to split a maildrop into two would require updating each registry's steerlist explicitly. There is no way for the underlying message forwarding system to determine automatically which messages for the split maildrop were to be delivered to which of its replacements.

One function that seems useful, but which we do not provide, is the ability to locate and replace all instances network-wide. The reason for this omission is that it is difficult to define the meaning of such a command. Not all nodes may be available at any given time. New nodes may be added while the command is in progress. New instances of the subsystem type in question may be added to those nodes prior to entry into the network. Thus, no accurate listing of the instances is attainable. Furthermore, if we then replace all those instances, we cannot assure that some new node entering the network does not have an instance of a subsystem satisfying the criteria for replacement. Theoretically, to allow this kind of command would require the ability to check every node entering the network for an indefinite period

of time into the future to ensure that no such instances existed. (It might not need to be forever, if we could ensure after some period of time that the type no longer existed or if the definition of the library ensured that a new node entering the network could not have access to such types.) While providing network-wide replacement is an important issue, it is not addressed here. The problem bears similarity to issues raised in [4].

5.6 Summary

We have identified a number of functions required for locating subsystems to be replaced, as well as for performing the actual replacements. In summary, performing a replacement requires not only the ability to substitute a new instance for an existing one, but also the ability to modify and transfer the state information between instances and to move data between nodes. This last requirement implies that it also must be possible to manipulate state objects at remote sites before or after moving them. Furthermore, users must be able to state the association between the handlers from replaced guardians, and those from newly installed guardians, so that communication with clients can continue uninterrupted.

Chapter Six

A Mechanism to Support Replacement

The previous chapter discussed the expressive power requirements for performing replacements of subsystem instances in Argus. In this chapter, we propose a mechanism to provide that power. We will examine not only the specific set of commands needed, but also the semantic and implementation issues that arise in defining those commands.

The description of the mechanism is divided into several parts. First, we describe the replacement system environment and namespace in which the replacement commands are invoked and executed. We then describe several sets of commands needed for performing single site replacements, including commands for: locating subsystem instances, creating and deleting guardian objects, transferring handlers, and transferring state values. We then discuss the handling of inter-site communication during replacement and performance of multi-site replacements. Lastly, we discuss the issues involved in providing replacement as an atomic action in the Argus system.

6.1 User Environment

Due to the complex nature of the tasks that must be performed during the course of replacement, the replacement system must support a fairly extensive user environment. There are several aspects of dynamic replacement that require special support.

In addition to the set of replacement commands that will be provided, the environment will need to support invocation of user-defined procedures for state transformation, as well as local variable declaration and assignment for storing intermediate values when complicated transformations require many steps. When a user enters the replacement system, a local environment is created. Each user has an independent environment. Initially the namespace in that environment contains only the Argus basic types and the replacement commands. The replacement system can access the library and install other types, given type and implementation information. Hence, other types and procedures can be loaded as needed. All types used in any guardian named for replacement are loaded automatically.

Since we are concerned here primarily with presenting the semantics of a dynamic replacement mechanism, we will not address the issues of providing a friendly user interface. A higher-level interface and full replacement language can be constructed using the basic commands and environment described here. In the examples given in this chapter, we use Argus syntax. An actual user interface for a replacement mechanism would probably be more interactive.

One necessary feature of the environment is that it be fully typechecked. Since we are performing replacement in a strongly typed system, we must be able to type check replacement functions and intermediate results to ensure type consistency at the conclusion of the replacement. However, the type checking we will be assuming in the specification of the commands must be performed dynamically (at the time of invocation) because it depends on the implementations, as well as the types, of the subsystems in use.

Because the function of the replacement system is to manage the changeover from one implementation of a type to another, types and multiple implementations of a type must be dealt with explicitly; the simultaneous use of multiple

implementations of a type is a necessity. The user environment therefore cannot be described as a set of Argus procedures, because types are not considered objects in Argus and hence cannot be manipulated as data. Furthermore, the Argus system does not deal with multiple implementations of a type in the same address space. The replacement mechanism must depend on the availability of symbol table and binding information about modules being replaced. Since a dynamic debugger [2] will require this information also, we will assume the information exists and can be accessed by the replacement system. When an invocation of a cluster operation occurs within the replacement environment, the appropriate implementation of the operation can be selected based on the implementations associated with the argument of appropriate type. If the operation has two arguments of the type, and they are of different implementations, the operation will fail.[8] If the implementation cannot be identified (for example if there are no arguments to the operation), the user will be asked to identify the implementation. When new guardians are created during replacement, the guardian_image to be used is provided explicitly by the user.

Thus, the replacement system provides an interactive user environment that allows invocation of Argus procedures and replacement commands, performs type checking and version management, and maintains a set of local objects and variables. In section 6.6, we discuss how this environment can be extended for multi-site replacements.

---

[8]It will be left to the user to coerce the two objects to the same implementation, for instance by using the encode and decode operations of the type.

6.2 Locating Guardian Instances

When a user enters the replacement system, the first task is to locate the subsystem instance to be replaced. As discussed in Section 3.3, we are unable to maintain instance information dynamically. Hence it is left to the user to locate all components of a subsystem at the time of replacement. Here we discuss the basic set of commands we provide to aid users in locating guardian and subsystem instances in the Argus network.

Perhaps the most direct way to locate guardians is through the catalog. As mentioned in Chapter 2, Argus provides a catalog of services available in the system. Handlers for those service subsystems are listed in the catalog under a "service name" (usually the DU name). Requests to the catalog will return the set of handlers for each subsystem instance associated with the service name supplied. (The catalog may allow other properties to be specified as well, for example location.)

To perform replacement, the user will need the unique identifier, or *gname*, for the instance, rather than its handlers. (This requirement stems from the fact that all of the handlers for a subsystem may not be connected to the same guardian.) The gname can be obtained via the command: get_gname(handler_id), which returns the gname of the guardian with which that handler is associated..

For those subsystems not listed in the catalog, the replacement system provides its own commands. The command:

all_instances(typename, impl_id, node_id) returns (seq[gname])

returns a sequence that contains the identities of all guardian instances of the given type and implementation that exist at the node given. The typename identifies the

DU in the library and any parameters, if the guardian type is parameterized. Hence to obtain the mail system's instances at the user's node, the following command would be used:

all_instances(mailertype, impl, home)

Note that since subsystems are not entities understood by Argus, the replacer can only find guardian instances, not subsystem instances. We therefore had to request instances of the root guardian for the mail system.

A problem arises here in that new instances of the subsystems in question may be created during or after a search is made for those subsystems. Therefore, by the time the user starts replacing instances, the information about existing ones may be inaccurate. (Instances may also be terminated but this presents less of a problem.) If all instances at a node are to be changed, we do not want new instances being created during replacement. The replacement system itself cannot control creation of new instances. It must depend upon the existence of an Argus system command that suspends the ability to instantiate (i.e. load) instances of a type. The user can then make the appropriate types or implementations unavailable for creation until after replacement. The existence of this command is one requirement that the replacement mechanism must place on other parts of the Argus system.

We have provided no command for locating all instances of a guardian type network-wide (only node-by-node) for a similar reason. We cannot control entry into the network of new nodes that can instantiate guardians of a type being replaced while replacement is taking place. We therefore require that users name each node explicitly, so that instantiation can be stopped at that node prior to replacement. We discuss this issue in Chapter 7.[9]

[9]This problem is similar to the "phantom" problem discussed in [4].

The user may also need to examine the state of a guardian under consideration for replacement, or possibly examine the state of client guardians to determine which instance of the service those clients are using.

To provide a mechanism to return a copy of a state object requires cooperation from the stable storage system. Without replacement, stable storage is read only during recovery, when the guardian is inactive. However, stable storage is never updated in place, so it should be possible to read the last copy even while a new copy is being written. We therefore can modify the stable storage interface to allow a request to read the last copy without locking the primary copy in volatile storage. The form of the command we provide is:

```
examine[t](gname, string) returns (t)
                          signals (wrong_type, no_such_variable)
```

where the string argument contains the name of a stable variable in the guardian named by the first argument and t is the type of the object named by that stable variable. Note that this command can be properly typechecked dynamically, but not statically. The type t depends on the type and implementation of the guardian named by gname. Neither of those pieces of information are available at compile time. Note also that if two consecutive *examines* are performed on the same guardian, they may not provide a consistent version of the state, since the guardian has continued to execute. Examine provides access to each stable variable independently, instead of copying the entire state, because we believe users will often need only part of the state, and copying all of a large state would introduce unnecessary overhead. However, for simplicity, we assume the entire state reachable from the stable variable will be copied. (Depending on the implementation of stable storage, it may be possible to optimize this, copying atomic objects only as they are used.)

It should be noted that the commands presented here depend upon certain information being available in the system at the time of replacement. This information includes the list of guardian instances currently existing at a node, along with the type and implementation associated with each guardian instance. The guardian management in Argus will keep the list of guardian instances along with the connection to stable state and to a copy of the load module for each. These are needed for recovery purposes. As mentioned earlier, the type and implementation information is kept for debugging, as well as replacement. How efficient commands like all_instances are depends heavily on how easily accessible this information is. Keeping the type and version information in the guardian manager's state rather than in each guardian will make implementation considerably easier.

6.3 Managing Guardian Instances during Replacement

This section discusses the functions required to manage collections of guardians during replacement. The user must be able to indicate which guardians are actually to be replaced. When a subsystem is being replaced, the subsystem root must be indicated so that appropriate checking can be performed. The command:

replace(typename, typename) signals(no_such_type)

states that a subsystem of the type specified by the first argument is to be replaced by an instance of the type given by the second argument. The second typename must be the same as the first, or the type it names must be a legal extension of it (section 4.1.3). As discussed in section 4.3, if the two are not identical, the guardian type may not appear in the interface of any of the handlers. The replacement mechanism will compare the subsys_handles clauses in the two DU's to ensure compatibility.

The replace command is intended to inform the replacement mechanism that the interface to the subsystem, rather than the interfaces to individual components, should be used for typechecking the replacement. The replacer will then allow handlers that are not identified in the *subsys_handles* list of the replaced type to be eliminated during replacement.

The replace command also has the effect of starting the replacement *transaction.* All actions performed between execution of the replace command and the end_replace command (described later) occur as a single atomic action with respect to clients of the replaced modules. Interruption of replacement by a node crash or explicit abort will cause all effects of the transaction to be discarded. It should be noted that the replacement commands described in the remainder of this chapter are not handlers, and their invocation will not initiate nested actions.

Because replacement requires exclusive access to the guardian instances, those instances will have to be removed from service before replacement can continue. This is done via the command:

remove(gname)

There are numerous options for scheduling this removal relative to client transactions requiring service. These will be discussed later in this chapter (Section 6.7). The guardian removed should be of a type named in a previous replace command or in the *includes* list of a previously removed guardian.

The remove command has the effect of terminating the guardian. By the semantics of guardian termination in Argus, the guardian is permanently destroyed only if the transaction that executed the terminate command commits. If the transaction is interrupted by a crash, the guardian is restarted. Though a guardian that has been

removed will appear to clients to be crashed, the Argus system must be able to differentiate between crashed and removed guardians for two reasons. One is that crashed guardians can be replaced, while guardians that have already been removed for replacement must be unavailable to future replacement transactions. The second reason for the distinction is that crashed guardians can be automatically restarted by the system, while removed guardians cannot be.

Once the guardians to be replaced have been identified, the new subsystem instances and their components must be created. This can be done using the function:

new[t](gimage) returns(gname, t)

The parameter *t* is the type of the guardian to be created, for example *mailer.* The object of type t returned is the guardian interface object of the newly created guardian; gname is the unique identifier of the new guardian. The gname returned could be derived from the interface object. However, both will be needed during replacement, and since Argus allows multiple return values from procedure invocations, returning both values will save an additional invocation. Gimage is the name of the guardian image in the DU for t.

New creates a new instance of the guardian, without running any creator. The module is loaded, and its handlers are created. No volatile or permanent state is written. It is the user's responsibility to write the permanent state. The replacement mechanism will ensure that all state components are initialized before allowing the replacement transaction to commit. As with creators, guardians instantiated via the *new* command will become recoverable upon commit of the transaction. If the replacement aborts, the newly created guardians will disappear. (Unlike creators, guardians created by *new* are not immediately available to clients.)

If a guardian has been removed via the replace command but is to be restarted without being replaced, a

restart(gname) signals (modified)

command is provided. This procedure may be necessary if some data is needed from the state and cannot be retrieved via the *examine* (perhaps because it is mutable). The effect of restart is to start the guardian as if it were recovering from a crash; hence the recovery code is invoked. Because there may be consistency constraints between these guardians and the replaced or new ones, restart will not take effect until commit of the transaction. Similarly, an explicit *terminate* command is used if for some reason a new guardian is not to be installed at the end of replacement.

Normally, all new guardians will be restarted automatically, and all removed guardians terminated at the end of replacement. The *end_replace* command performs this function. *End_replace* is invoked by the user to inform the replacer that the construction of the new subsystem is finished, and that subsystem may be started in place of the one removed from service. The status of the removed guardians must then be changed to *terminated*, so that they will be permanently destroyed upon commit of the replacement transaction. The Argus system must be informed that the new guardians have "crashed", so that they will then be "recovered" when replacement commits (thereby retrieving the newly created state from stable storage, and initializing volatile state.)

End_replace will refuse to commit the transaction if certain conditions on the state and handlers of the guardians have not been met. We discuss those conditions is sections 6.4 and 6.5.

The counterpart to the end_replace command is *abort_replace.* In this case, the replacement mechanism ensures that all guardian instances revert to their status prior to the replacement. New guardians are destroyed. Removed guardians will resume execution as if recovering from a crash (unless they were in a crashed state at the start of replacement, in which case they remain crashed.)

6.4 Continuity of Communication

One primary criterion we have imposed on the replacement system is that clients need not be notified of a replacement. To avoid such notification, messages for handlers of replaced guardians must be forwarded to the corresponding new handlers automatically. The user performing the replacement must explicitly define the mapping from old to new handlers using the command:

rebind[structtype] (gname, structtype) signals (incomplete, bad_handler)

where gname is the name of the replaced guardian, and the structtype is either a struct[handlers] or a guardian interface object. The struct[handlers] must have fieldnames and components corresponding to the handler names and handler types in the old guardian. The component values are the handlers to which the corresponding old handlers are to be rebound. It need not contain all the handlers from the old guardian, if some of those handlers are not in the *subsys_handles* list.

The fieldnames in the structtype are used to determine which handlers in the old guardian are to be rebound. A handler in the old guardian will be rebound to the component of the struct[handlers] with the fieldname corresponding to the handler's name. Old handlers with no corresponding fieldname in the struct[handler] will not be rebound. Rebind will signal *incomplete* and the rebinding will not be performed if such a handler is in the subsys_handles list. Internal handler names may appear in

the struct[handlers]. The names of new handlers for extended types need not appear, as they have no old handlers rebound to them. Only handler names that appear in the subsys_handles clause of the old guardian may be in the struct[handlers]; otherwise, *bad_handler* is signalled.

The second argument can also be an interface object; there is a simple correspondence between a system-defined guardian interface type and a struct[handler]. This option is provided because it is far easier to describe single guardian replacements using the interface type directly. As with the basic struct[handler], the handler types in the interface object must correspond to handlers in the old guardian, and all old handlers covered by the subsys_handles clause of the subsystem must be rebound. The rebind command must be executed exactly once for each removed guardian of the type named in the replace command, and for each guardian named in the subsys_handles clause of a guardian that has been rebound. Thus, if a mailer named old_mailer is to be replaced by a new mailer instance, the following commands might be used:

mailername:gname, new_mailer:mailer := new[mailer](mailer_image)
rebind[mailer](old_mailer, new_mailer)

The asymmetry in the rebind command exists for the following reason. All of the handlers of each removed guardian must be explicitly rebound, except for those handlers internal to the subsystem that are not used by the new implementation. On the other hand, the handlers bound to a new guardian may come from several different replaced guardians. By insisting that all of the old guardian's handlers are rebound at once, we make it less likely that the user will forget to rebind one. Furthermore, a check can be performed at the time of the rebind to ensure that all handlers from the guardian that are in the subsystem interface are being rebound.

A problem arises in using rebind when handlers not in the *handles* list of the guardian are nonetheless passed to clients. Since these internal handlers are in the guardian's interface, they must be rebound; however, they are not in the guardian's interface object and are not returned by the get_handlers operation. An operation

get_handler[t](gname) returns (t) signals (no_such_handler)

is therefore needed to retrieve the handlers. Here, t is the handler type. The handler must be explicitly included in the set of handlers comprising the second argument to rebind. The internal handler of that name will be automatically retrieved from the guardian named as the first argument of rebind. Note that this problem arises only with handlers that are internal to a guardian. Handlers that are in the *handles* list of any replaced guardian can be accessed without this special command.

Any number of existing handlers may be rebound to the same handler. Thus, in our example, if three maildrops were to be merged, they could all be rebound to the same handler, by performing three rebinds with the same second argument.

The handler_id of the destination handler remains distinct from that of any of the handlers rebound to it. If equal(handler1, handler2) was false before replacement, it will remain so after replacement, even if handler1 and handler2 are rebound to the same handler. Because of the difference in handler_ids across a replacement, there will be cases in which the occurrence of a replacement is visible to clients of the replaced subsystem. These cases arise when a client acquires the id for the same handler twice. If the guardian to which that handler belongs is replaced between the two acquisitions, a comparison of the two acquired identifiers will reveal the replacement. If no replacement had occurred, the id acquired each time would have been the same, whereas an intervening replacement will cause a change in the id. In

the standard case, clients invoke handlers but never examine the handler_id; hence, replacement is not visible. However, as long as clients have access to an equal operation on handlers, it is not possible to completely hide the occurrence of a replacement. (Note that this situation is not evident in our formal model, as the distinction between handler id and handler definition is not made.)

The implementation of rebind is dependent on the implementation of the underlying message distribution system. Rebind cannot be performed unless a level of indirection exists between the handler name given to clients and the handler address. The Argus message transmission mechanism must be able to redirect messages. As part of a replacement, this message distribution subsystem must be notified of the change in location.

6.5 State Management

6.5.1 State Access Procedures

Once a guardian has been removed from service, the user may access its permanent state via the state manipulation commands. For the single node case, when state information is not being transferred between nodes, we have two basic commands:

get_state[rt](gname) returns (rt) signals (wrong_state_type)

and

put_state[rt](gname, rt) signals (wrong_state_type)

In these interface specifications, rt is the record type corresponding to the state object of gname. (Specifically, the field names of rt correspond to the stable

variable names in the state of gname, and the component types correspond to the types of the objects associated with those stable variables.)

Since the state transfer cannot be performed automatically, the put_state must be explicit even when no change has occurred. For those cases in which the state must be transformed during replacement, some translation operations will be called between the get and put operations. It is for this reason that the replacer's user environment must allow arbitrary procedure calls.

The type correctness of the get_state and put_state operations cannot be determined by static typechecking. Whether rt corresponds to the state type of gname depends not just on the guardian type of gname, but on the implementation instantiated. The replacer must check the implementation of gname explicitly, and retrieve the state type information from the library or debugger's database.

A put_state operation must be invoked for every guardian instantiated during replacement (that is not explicitly terminated prior to end_replace). End_replace will not permit the transaction to commit until this condition is satisfied.

We have provided only operations to access and store the entire stable state of a guardian, because it seems likely that state transfer during replacement will require use of the entire state value. There is no difficulty associated with providing operations to access individual state components (as demonstrated by the *examine* command). Such component access operations could be included in a more extensive user environment. In addition, the implementation of these operations can be optimized to retrieve atomic objects from stable storage only as they are used, and possibly, to avoid copying the state from stable storage at all, if it is being transferred directly from the old to the new guardian, without any translation. How this optimization can be performed, and how much can be optimized depends

extensively on the implementation of stable storage. We therefore cannot provide the details of such optimizations here.

6.5.2 Implementation Issues in State Management

The major problem related to state manipulation is performing replacements when transactions have *prepared* data in the guardian. Guardians might be removed from service when prepared transactions still hold locks. Therefore, the replacer must have the ability to handle the case in which some part of the state is *prepared* at the time of the get_state operation.

A number of approaches to handling prepared state are possible. The simplest is to refuse get_state operations until the transactions involving the state complete. Note that stable storage is still accessible to the transaction system, though the guardian has been removed from service. Hence, the transactions will eventually commit or abort. If the replacement system refuses the get_state, rather than blocking it until the transactions finish, users can continue with other parts of the replacement.

An alternative method of handling prepared transactions is selection of a scheme that accomplishes as much of the replacement as possible before the prepared transactions complete. One such scheme would translate the guardian state once for each combination of possible commits or aborts of prepared transactions. If only one prepared transaction existed, then two translations would be required. If a long delay in resolving the transaction state were expected, such an approach would allow replacement to complete as soon as the transaction finished, instead of having to perform the translation at that time. This approach is derived from [Montgomery]. The solution is exponential in the number of prepared transactions. Whether it actually reduces delay depends upon the average number of prepared transactions expected at a guardian at any given time, the average delay in the second phase of

commit, and the amount of processing power available to perform state translations while waiting for the transactions to resolve. A detailed performance evaluation would be required to determine the efficacy of this scheme. The technique might prove useful in appropriate circumstances; we do not have the performance data available to determine its usefulness at this time.

The following is a less expensive solution that reduces delay in most but not all cases. (It is based on the assumption that most atomic actions, once prepared, will commit; if this assumption is false the system will make no progress.) This algorithm will reduce the delay in returning replaced guardians to service when the transactions that prepared the data commit. Rather than translating the state for all possible combinations of commits and aborts of outstanding transactions, we make the optimistic assumption that all of those transactions will commit. In response to a get_state command, the replacement system retrieves the state values that will result if the transactions commit, and performs the translation on these values. The replacement transaction will not commit until the prepared transactions resolve. If any one of those transactions aborts, the replacement transaction will have to abort. Thus, if prepared transactions usually commit, we can reduce the time guardians are out of service for commit, but we risk having to redo part of the replacement if a prepared transaction aborts.

6.6 Multi-Site Replacements

As discussed in Section 5.2.3, management of multi-site replacements involves directing the manipulation of state objects and invocation of replacement commands at remote nodes. Due to nontransmissibility and type incompatibility (as defined in section 5.2.3), state values cannot always be moved to the user's node without first performing a translation at the remote site. In the worst case,

complicated replacements may involve relocating data from one remote site to another. Here, we examine the facilities users will need to direct such replacements.

6.6.1 Remote Replacement Commands

Some of the replacement functions described thus far must deal with multiple sites as part of their function. For example, since a single rebind command is used for all of the handlers of a removed guardian, it must be able to perform the rebinding across nodes automatically: not all of the handlers of a removed guardian will necessarily be rebound to handlers at the same site. Similarly, we will assume the *replace* command need be executed only once, although components of the subsystem type specified may span the network. The all_instances command explicitly takes a node_id as an argument. All other commands, however, involve only a single node. In this section we describe a multi-site user environment that permits invocation of replacement functions at remote sites.

Users must be able to perform the standard replacement commands, such as remove, get_state, put_state, and new, at remote sites. In particular, get_state returns a value; that value cannot be returned to the user, since it may not be transmissible. Therefore, the user must be able to state where the return value is to be stored. In a complicated transformation, it may be necessary to invoke a series of procedures at a remote site. The arguments to those procedures may be results of previous commands at that site, or values received from other sites participating in the replacement. For example, if we are combining the maildrop at node A with the maildrop at node B, then the state of maildropA must be sent to node B, and a procedure invoked there to combine the states of the two guardians. The user may be controlling this replacement from node C.

We therefore need a mechanism that allows variables to be declared and assigned

values at remote sites, as well as allowing procedure invocation with those local variables as the arguments and targets for return values. Finally, the mechanism must support transfer of abstract values between nodes.

We propose the following mechanism. In section 6.1, we described a local replacement environment. We now propose that that environment be extensible to a distributed environment with parts at remote sites as well. While we would like a single unified environment, for the purposes of our description here, we make one simplifying assumption. To avoid the problem of a distributed namespace with globally unique user-defined identifiers, we will assume that names are unique only to their local site. Variables may be assigned to or used as argument to invocations only at the site at which the variable is declared. Values must be explicitly moved between sites.

The user's site serves as "coordinator" of the replacement. It maintains a list of all guardians removed and created during the replacement, along with the sites of those guardians. The replacement mechanism at each site must transmit to the coordinator information about replace, remove, and new commands executed at remote sites. Hence, the replacement mechanism has the information to type check across nodes (for example, to ensure that all removed guardians are in the closure of the root's includes list.) End_replace will check with the replacement system at each participating node to ensure that all handlers in the subsystem interface have been rebound, that all new guardians' states have been initialized, etc. Movement of values between sites will have to be explicit.

We present here a minimal set of commands to direct actions in these remote environments, and move values between them. First, we provide a command to enter a remote environment. Once opened, an environment exists for the duration of the replacement. The command is:

connect(node_id)

If the user already has an environment at that site, the replacement system reconnects to that environment. If no such environment exists at the named site, one is created. Functionally, *connect* causes all statements entered within its scope to be executed within the replacement system at node_id. The only objects that may be named, for example as arguments to procedure invocations, are those assigned to variables declared in that local environment.

End_connect leaves the connect scope, and returns to the environment that was in use prior to connect. Connects may be nested.

To access a variable declared at another site, the command

```
retrieve[t](node_id, string) returns (t)
                              signals(no_such_variable,
                              non_transmissible)
```

is used. The string argument must contain the name of a variable in the user's environment at the node named by the first argument. The command is executed from the site to which the value of variable name is to be moved. Only values stored in local variables at node_id can be retrieved. Therefore, if a state value is to be moved, the user will have to connect to its current site, perform the get_state there, then return to the destination site, and execute the retrieve command. (Obviously, a higher-level user interface could provide a single operation that retrieved a transmissible state value.) If, for example, a user at node C wanted to transfer the list of mailboxes in a maildrop state from node A to node B, the following command sequence would be needed (assume maildropstate = record[boxes:box_list]):

```
replace(mailer, mailer)          %no change in type
connect(node_A)                  %state is at node_A
  remove(maildropA)              %stop guardian and
                                 %then retrieve stable state
```

```
  boxes: box_list := get_state[maildropstate](maildropA).boxes
  end_connect
connect(node_B)                          %want to move value to node_B
  boxes:box_list: = retrieve[ box_list](node_A, "boxes")
  %perform rest of replacement
  end_connect
```

Note that the variable names need not be unique across nodes. Thus, the connect scope, and retrieve commands provide the minimal function necessary to control multi-site replacements.

6.6.2 State Relocation

We now have a replacement environment that provides a way for the user to connect to remote nodes and perform replacement tasks there. In addition, the retrieve command allows the user to move objects among the various sites to support guardian relocation. There are however, two problems the user faces during the course of such replacements for which the replacer can provide only minimal support: type incompatibility and nontransmissibility (see section 5.2.3). In either of these cases it will be the user's responsibility to convert the data to a form that can be relocated. This section discusses the actions required to perform a multi-node replacement, and the structure imposed on such replacements by our choice of primitive internode operations.

There are several possible ways to move objects that are not transmissible to a new location. One is to transform the objects into objects of types that are transmissible. If the Argus library supports type extension for clusters, the object can be coerced to an extended type that provides encode and decode. If data type extension is not directly supported, the user will need access to the object's representation (rep), i.e. the objects of lower-level types used to define the abstract type. In Argus, the operations *up* and *down* that convert between the rep type and abstract type are allowed inside cluster operations only. We provide the analogous operations:

up[rep, abstract](rep) returns (abstract) signals (bad_type)
down[rep, abstract](abstract) returns (rep) signals (bad_type)

The rep type itself may be transmissible, in which case the user simply needs to invoke *down*, relocate the object, then invoke *up*. If the same representation is not used at both sites, some translation may be needed before invoking *up*. *Down* can be invoked repeatedly to get to lower-level types. If the rep is not transmissible, an alternative is to *down* the object, then use *up* to coerce it to an extended type with encode and decode operations.

Use of these operations will be checked to ensure that the rep and abstract type match in the implementation of the type used for that argument, and that the argument is of appropriate type. Obviously, the existence of these commands again provides the opportunity for misuse of the replacement mechanism.

One special case exists in which nontransmissible objects can be moved to a new node without user intervention. If the same version of the type exists at both nodes and the processors are the same type, then the base representations of the objects are identical at both locations. In this case, the replacement mechanism could transfer the object using the copy operation used by the garbage collector.[10] (See [MPH] for a discussion of using this method for transmitting abstract objects.) Whether this case is common enough to warrant the special implementation will not be clear without data on usage patterns.

The above solutions will work as long as the types of the objects in the old state exist at the new location, regardless of whether those types are actually used in the new version. If the types used in the new state are available at the old node, the state

---

[10]This operation is known as gcdump.

transformation can be performed before transmission. Otherwise, some type common to both nodes will have to be found in which to represent the state information.

6.7 Scheduling Replacement

The issue of scheduling primarily involves the synchronization of *remove* commands relative to clients' handler invocations. At any given time, clients may be running handlers of the guardians, or some of the guardian's state objects may be locked by transactions that have not yet completed. The most straightforward method of removing a guardian for replacement would be to gain exclusive access to it via a standard locking mechanism. This method requires exclusive and shared locks on guardian objects. Replacement would be scheduled by requesting an exclusive lock on the guardian and waiting in a queue, along with other actions requesting guardian access, until the lock arbitrator granted the lock request. Handler invocations would then have to acquire shared locks. If scheduling of replacements could be performed in this manner, no additional scheduling mechanism would be needed specifically for replacement. This simplicity in both semantics and implementation would be a strong advantage. In addition, this method has the advantage of minimizing disruption to the rest of the system, since it uses standard system priorities.

However, this method also has several disadvantages. First, the locking strategy currently being used in Argus would have to be modified. While locks exist for guardian objects, they are used only to prevent conflicting writes (e.g. termination before creation commits). No lock is required for invoking a handler. Thus, a change in language semantics to require such locks would be needed to use this

scheme.[11]

Another disadvantage is that a human user is controlling replacement interactively and that user will be forced to wait while the replacer acquires all the needed locks. The length of the wait will depend, not only on how busy the guardian is, but on how the Argus locking scheme arbitrates requests. Another disadvantage is that replacements often involve many guardians. If some guardians are acquired immediately, and it is necessary to wait for others, those acquired first will be out of service for a longer period of time, thus increasing disruption in service.

The possibility of deadlocks between replacement and other transactions using the guardian is yet another issue. A transaction might first acquire a read lock for guardian A, and then request one for guardian B, while the replacer gets a write lock on guardian B and then tries to obtain one for guardian A. Thus, the replacement system can introduce deadlock into a subsystem that may have been proven to be deadlock free under normal operation. This presents semantic as well as performance problems.

An alternative is to have replacement preempt transactions using the guardian. In this case, the remove command will cause all handler executions in progress to be aborted. The guardian will be made inactive: its processes are killed and messages to its handlers will be blocked. Furthermore, all read and write locks on the guardian's state that are held by unprepared transactions must be broken. Only locks from "prepared" transactions remain. Therefore, all unprepared actions that held locks will be aborted.

---

[11]There were a number of reasons for the decision made regarding the locking protocol for handler invocation. It is beyond the scope of this thesis to evaluate the tradeoffs between those reasons and the requirements cited here.

This preemption method has the advantage of giving priority to human users; the person performing the replacement will not have to wait inordinate amounts of time for each of the guardians being replaced to be seized by the replacer. In addition, since each guardian is removed from service as soon as requested, the average time during which a guardian is unavailable to clients is smaller than in the previous case.

The best method of scheduling of replacements therefore depends in part on the characteristics of the particular system. If it is important that running transactions not be aborted, the first alternative might be considered superior. The decision on which to use could be left for the user to decide at the time of the replacement, so that characteristics of the individual subsystem being replaced may be taken into account. In a system with varying subsystem characteristics, or in which the necessary performance information is unavailable, the second option, preemption, appears to be the best alternative. It will on average cause the least disruption to the system as a whole, and the best service to users performing replacement.

6.8 An Example

In the previous chapter we gave an example of a restructured mail system that used only mailer and mailsite guardians. Here, we show how an instance of our original mail system could be replaced with an instance of the new structure. Suppose the old instance is configured as in Figure 6-1. Assume that only nodeA and nodeB are in the network. The mailers can be found through the catalog, but the identity of the registries and maildrops must be found through a mailer's state. There is a registry at nodeB, and maildrops at nodeB and nodeA. The new instance is to have a mailsite at nodeA only, and mailers at each site.

The first step is to obtain the mailers from the catalog. From the mailer's state, the

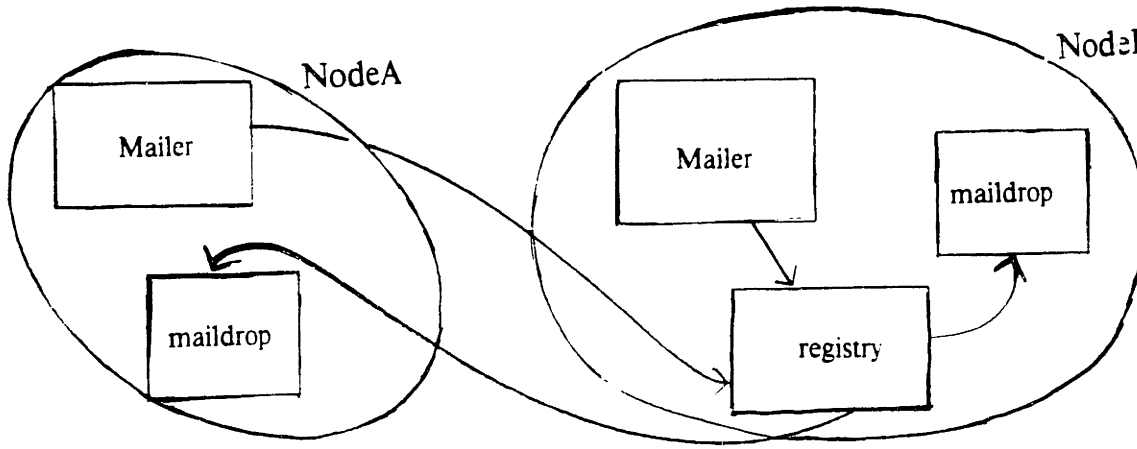

Figure 6-1: The Mail System Configuration

registry named by *some* is obtained. With access to a registry we can locate all of the maildrops and other registries. The following commands will start the replacement of the mail system, remove a mailer, and locate the registry named in the mailer's state. We assume the user has access to the node names at the start of this replacement session.

```
rstate = record[regs:reg_list, steers:steer_list]      %type abbreviations
mstate = record[some:registry]
dstate = record[boxes:box_list]
sitestate = dstate
newmailstate = record[site:mailsite]

replace(mailer, mailer)                                %inform system of type being replaced

mailerA:mailer := catalog.find[mailer](nodeA)          %find the mailers
mailerB:mailer := catalog.find[mailer]@(nodeB)

Remove(mailerA)                                        %get one mailer
r:registry := get_state[mstate](mailerA).some          %get registry from state
rname:gname := get_gname(r.lookup)                     %pick any handler
rloc:node_id := get_node(rname)                        %find registry's node; note: rloc = nodeB
```

The next step is to connect to the node at which the registry exists and retrieve its stable state. (Note that the user can determine at this time that the registry is at

nodeB, and the commands executed after this point make use of this fact.) Since the user does not know how many other registries or maildrops there are, the commands are written to loop over any number of registries and maildrops. First, the list of users and their mailbox locations (the steerlist) is searched to find all the maildrops. We then test whether each drop is at nodeB or nodeA. If is at nodeB, it is removed; its mailboxes are retrieved and appended to a list of all mailboxes. (We assume the *concat* procedure has been loaded.) If the drop is from nodeA, its name is recorded, so that nodeA can be provided with a list of the drops to be accessed. This code takes advantage of the knowledge that only two nodes exist. The next step is to iterate over the list of registries, removing them if they are at nodeB, recording their names if from nodeA. The state of these registries need not be accessed, since registry states are redundant and we have already obtained one copy.

```
connect(nodeB)                                  %connect to registry's node
 rname:gname := retrieve(nodeA, "rname")        %get the registry's gname from nodeA
 remove(rname)                                  %remove registry from service.
 state:rstate := get_state[state](rname)         %get the state
 allboxes:box_list := box_list$create()         %array for storing maildrop state
 ADrops:array[gname] := array[gname]$create()   %maildrops from other node

 for st:steering in steer_list$elements(state.steers) do   %look for all maildrops
  drop:gname := get_gname(st.drop.read_mail)    %get gname from interface object
  dnode:node_id := get_node(drop)               %find location of drop
  if dnode = nodeB                              %if its here then
   then remove(drop)                            %remove it from service and
      concat[box_list](allboxes, get_state[dstate](drop).boxes) %add mailboxes to common list.
   else array[gname]$addh(ADrops, drop)         %if not here, save name for nodeA
   end %if
  end %for

 ARegs:array[gname] := array[gname]$create()    %list of regs at other node
 for nextreg:registry in reg_list$elements(state.regs) do   %look for all registries
   nrname:gname := get_gname(nextreg.lookup)    %get gname from interface object
   if get_node(nrname) ~= nodeB                 %if reg isn't at this node then
    then array[gname]$addh(ARegs, nrname) %save name in list for nodeA
    elseif reg ~= rname then remove(nrname)     %if not yet removed, remove it
    end %if
  end %for
 end_connect
```

Control is then returned to nodeA. At nodeA, the user must obtain the values computed at nodeB: the list of mailboxes retrieved thus far, the list of nodeA's registries, and the list of nodeA's maildrops. The second list is empty. The user then removes the maildrop, and concatenates its list of mailboxes to the existing list.

```
%back at NodeA

ARegs:array[gname] := retrieve(nodeB, "aregs")          %get list of nodeA's regs

allboxes:box_list := retrieve(nodeB, "allboxes")
ADrops:array[gname] := retrieve(nodeB, "ADrops")

for nrname:gname in array[gname]$elements(ARegs) do     %in this case, ARegs is empty
  remove(nrname)                                        %Don't need other registries; remove them
  end %for

for d:gname in array[gname]$elements(ADrops)            %ADrops contains one maildrop
  remove(d)                                             %remove it
  concat[box_list](allboxes, get_state[dstate](d).boxes) %combine box lists
  end %for
```

The final step is to create the new guardians. A mailsite is created; its state will be the list of boxes from all the maildrops. A mailer is then created at each node; the state of each is a pointer to the new mailsite. Replacement is then completed.

```
sitename:gname, site:mailsite := new[mailsite](siteimage)  %create new mailsite at nodeA
put_state[sitestate](sitename, sitestate${boxes:allboxes})  %initialize state
                                                           %create new mailer for nodeA
mname:gname, new_mailer:mailer := new[mailer](image_using_mailsite)
put_state[newmailstate](newmailstate${site:site})

%create new mailer at nodeB
connect(nodeB)
  site: mailsite := retrieve(nodeA, "site")
  mname:gname, new_mailer:mailer := new[mailer](image_using_mailsite)
  put_state[newmailstate](newmailstate${site:site})
  end_connect

end_replace
```

6.9 Conclusions

This chapter presents the basic set of commands needed to accomplish subsystem replacements in Argus. The set of commands itself is small and reasonably straightforward. These include a statement to define the type of the subsystem being replaced, a command to remove guardians from service, commands to retrieve and install stable state, and end replacement. The complex part of the replacement system is the user environment it must support.

One principle difficulty in designing the replacement mechanism stems from the need to handle types and implementations of types explicitly. Since types are not objects in Argus, and multiple implementations are entirely a library notion, the mechanism cannot be defined in Argus terms, and replacement cannot be run as Argus procedures.

The commands presented in this chapter are sufficient to express the replacements necessary, and to describe the semantics of replacement in the Argus system. However, it is clear from examination of the example in the previous section that this minimal mechanism is insufficient as a user-level service. Replacement is a complicated process for which it is difficult to ensure correctness. It is therefore important to develop a methodology for performing replacements, and a language to support that methodology, that will provide the user with a higher-level view of replacement and aid in expressing the replacement tasks accurately.

The need for another addition should be evident as well. While we have provided the mechanism to replace a single subsystem instance, we have provided no additional support should the user wish to replace all instances of a subsystem or version of a given subsystem type. Obviously, such a task could be performed by repeating the set of commands explicitly for each instance. In an actual user

environment, however, there should be support for writing "replacement procedures", or "command files" that can be invoked repeatedly for different instances.

Chapter Seven

Conclusions

This dissertation has examined a number of aspects of the problem of dynamic modification of software modules in a distributed system. Our research focused on modules with long term, on-line state. The approach taken was to incorporate a facility in the programming system, but not in the language, to allow replacement of running modules with new implementations.

7.1 Summary and Analysis

We have identified the appropriate unit of replacement in Argus as the *subsystem*, a unit not recognized in the language. The principal reason for this decision was that choosing the smaller unit of *guardian* would not have provided sufficient power for many kinds of replacements we believe to be useful. Among the categories of modifications that would have been omitted are changes in the module structure of abstraction implementations involving more than one guardian. Without recognizing subsystems as a unit of replacement, a new subsystem implementation, which makes use of a different set of component guardian types than an existing instance of the subsystem could not be substituted for that existing instance. The problem is of particular concern because Argus recognizes no module types that cross node boundaries; hence, for services that span nodes, only multi-guardian implementations are possible. If subsystem entities were not recognized, the implementor would not have the power to create implementations that were easily replaceable.

Recognizing subsystems as entities without changing the Argus language, and without compromising the type safety of the language, necessitates extending the function of the library. Sufficient information can be added to the library to allow the replacement mechanism to handle most of the modifications we would like to support. There are still weaknesses in both power and safety, primarily due to the inability to identify subsystem instances dynamically; however, many type errors can be caught with our library scheme.

We encountered another limitation in the module structure of Argus that we were unable to rectify without language support. The mail system example used here was derived from an example in [12]. The original example allowed very little modification because the underlying component structure was visible to clients: the commands to add maildrops and registries appeared in the mailer's interface. It was, in fact, difficult to change that example to one in which the module structure was hidden from clients. The implication of this fact is that it may be difficult to create subsystems that are modifiable, either statically or dynamically. The weakness appears to be that, in reality, there are different classes of clients using different subsets of the subsystem interface, but the Argus type mechanism cannot recognize that fact. In the original mail system, it is likely that the clients using create, add-mailer, add-maildrop, and add-registry, are different from those using send-mail, read-mail and add-user. There is no way the replacement mechanism could allow changes to one subset of the interface, along with its clients, without affecting the other clients (and still guarantee type-safety). This ability would depend on language support for multiple interfaces to a type. (We discuss this possibility briefly in the following section.)

Chapter 4 examined the conditions for correct replacement of subsystems. Those conditions indicate that performing replacements dynamically will be more complex than expected. One result shown in this chapter is that there may be two

implementations associated with the same abstraction, both of which correctly implement the specification of the abstraction, and yet it may be impossible to substitute the second for the first dynamically. Either implementation could of course be used to create new instances. This situation arises when the specification allows nondeterminism. Conversely, it may be possible for one subsystem instance to correctly replace another when that instance could not be used to create new instances of the subsystem type. In this case, the new instance does not satisfy the specification that the original instance does; however it satisfies a *continuation specification*. Chapter 4 also analyzes the conditions under which a subsystem instance can be replaced by an instance of an *extension* of the subsystem type.

These results do not apply uniquely to Argus, but rather to any strongly typed language. The notion of data type has been a static one; static type checking, and the association of a *type* with a static module definition (for example, in the Argus library) places limitations on dynamic modifiability that are unrelated to the constraints necessary to ensure continuity of behavior. Section 4.3 shows, for example, that in many cases replacement of one abstraction by another in Argus will violate Argus type checking rules, though correctness conditions could be satisfied if abstractions were not statically associated with a type name in the library. Similarly, specifications for abstract data types have considered only modules that are assumed to start in a specific initial state, and whose behavior is defined from that state. If continuously-running subsystems are to be permitted to evolve gracefully over time, we will have to reexamine the static notion of type definitions in programming languages.

Finally, we have described the functional requirements of a mechanism to perform dynamic replacement in Argus, and discussed the semantics of such a mechanism. The mechanism defined is powerful enough to allow the desired range of replacements and perform the type checking possible given the constraints of the

library support described in Chapter 3. However, a higher-level user interface is needed. While the mechanism serves to illustrate how dynamic replacement can be implemented, and gives a more concrete form to the requirements outlined in Chapter 5, the operations provided are at too low a level to be easy to use, or to support a methodology for performing replacements.

7.2 Future Work

There are several research directions that may follow from this work. One path is extension of the mechanism; we have already mentioned several areas in which our mechanism is limited. Another area requiring further exploration is the development of a testing mechanism for the replacement system. Finally, several language design issues have come to light as a result of this research, and these require further investigation.

There are several possible extensions to the replacement mechanism defined in Chapter 6. One area that requires further work is the design of a higher-level user interface. In particular, such an interface should provide more structure for organizing the replacement task, and provide better support for replacement of multiple instances of a subsystem type, with such facilities as command files. Another area in which the mechanism could be extended is in recording the history of replacements. We showed in Chapter 4 that successive replacements could restrict the changes possible in the future. We can limit this effect by retaining information about the replacement history. Some obvious information to retain is the sequence of implementations that preceded the current instance, along with the final state of each of those instances (and possible the starting state). How useful such information is in increasing the flexibility of replacement must be evaluated. Furthermore, since the user performing replacement must be able to interpret that

information in constructing the new instance's state, the form in which that information should be kept must also be determined. Network-wide replacement is another extension that should be investigated. Finally, we noted earlier that our mechanism can only support replacement of guardian-based subsystems. Mechanisms to support more general subsystem replacement should be studied.

Another area still requiring study is a testing mechanism to use in conjunction with dynamic replacement. There are numerous ways in which errors can occur during the course of replacement. Of particular concern, however, is the user-defined process for translating between state representations. This process is probably the most complicated task required during replacement. If an error occurs during the translation, state inconsistencies can result. Hence, a method of testing the translation procedure to ensure that the state values before and after translation are "equivalent" would be useful.

Given the transaction mechanism available in Argus, several approaches suggest themselves. We believe it worth exploring a mechanism that allowed test transactions to be run in parallel on the old and new implementations. This approach would allow the user to test the translation procedure in conjunction with the replaced and replacing implementations to ensure that the procedure is generating a state that will produce proper behavior in the new implementation. Recall from Chapter 4 that state equivalence is relative to the two implementations (because it depends upon the abstraction functions of the implementations). It might even be possible to run the test transactions in the actual running system, to test interaction with other subsystems. The transactions could then be aborted to prevent permanent effects on the system.

Another approach to dynamic replacement that deserves examination is the incorporation of support for replacement into the programming language. In Argus,

this approach would require adding subsystems to the language, as well as adding specific support for replacing subsystems. The impact of these changes on the rest of the language would have to be studied as well.

There are a number of choices for including subsystems in the language. The basic features we need from such a mechanism are the following. First, clients should see only subsystem interfaces, not guardian interfaces. A single guardian is a special case of a subsystem; however, clients should not expect all handlers derived from an interface object to reside at the same guardian, or even at the same node. It should be possible to provide automatically created interface objects that contain handlers attached to several guardians. If such subsystem support were provided, it would eliminate the need to maintain the root guardian type during replacement, thus increasing the range of allowable replacements.

Another possible form of language support would be constructs to allow a subsystem to prepare for its own replacement. One approach to examine is whether a canonical state representation can be introduced, along with user-defined operations to translate between the local representation and the canonical form. This approach is analogous to the Argus approach to abstract value transmission, in which transmissible types must have a canonical representation and user-defined encode and decode operations. This method allows abstract data values to be transmitted between guardians using different implementations of the data type. Here we wish to transmit the subsystem state value between two implementations of the subsystem. We believe that incorporating support for dynamic replacement in the language, rather than just in the system, will lead to a higher-level, less ad hoc replacement mechanism. It also has the advantage of returning to the module control over direct access to the module's state, whereas in the current mechanism, a process outside the module is gaining direct access to the module's private state.

We mentioned earlier in this chapter that the single, unified view of a module interface did not accurately model usage of subsystem modules. One possible solution worth investigating is the notion of *multiple views* of a data type. This approach would allow different subsets of a type's operations to be declared as different views, with clients binding to a view of the type, rather than the entire interface. The work on multiple views of databases is probably relevant here [3, 1]. This structure would allow greater flexibility in modifying subsystems, while hiding changes from clients that are not affected by those changes.

Finally, we have noted conflicts between the requirements of dynamic modifiability and those of static type definitions, as they currently exist in strongly typed languages. We believe the meaning of types in these languages, and the methods used for defining types and performing static type checking, should be re-evaluated in the context of dynamic replacement. The meaning of correctness of a type definition, and specification methods for abstract types must also be extended to allow for dynamic updates.

In conclusion, as continuously running software with long-term state becomes more common, the importance of supporting modifiability for these systems will increase. This dissertation represents a first step in providing support for *evolving systems*, in which software is expected to change gradually over time, without disruption to the system; further support will require integration of the notion of dynamically changing modules into the languages in which those modules are defined.

References

1. Chamberlain, D., J. Gray, and I. Traiger. Versions, Authorization, and Locking in a Relationship Database System. *Proceedings of the National Computer Conference* (May 1975).

2. Chiu, Sheng-Yang. Debugging Distributed Computations in a Nested Transaction System. Ph.D. Thesis, M.I.T. Laboratory for Computer Science, forthcoming

3. Dayal, U. On the Updatability of Relational Views. *Proceedings of the International Conference on Very Large Databases* (September 1978).

4. Eswaran, K.P., J.N. Gray,R.A. Lorie, and I.L. Traiger. On the Notions of Consistency and Predicate Locks in a Relational Database System. *Communications ACM* (November 1976).

5. Fabry, R. How to Design A System In Which Modules Can Be Changed On The Fly. Proceedings of the Second International Conference on Software Engineering, IEEE, October, 1976.

6. Gray, J.N. *Lecture Notes in Computer Science. Vol. 60: Notes on Data Base Operating Systems.* Springer-Verlag, Berlin, 1978.

7. Habermann, N. Dynamically Modifiable Distributed Systems. Proceedings of the Distributed Sensor Net Workshop, Carnegie-Mellon University, Pittsburgh PA., December, 1978.

8. Henderson, Cecilia. Locating Migratory Objects in an Internet. Master Th., MIT Laboratory for Computer Science, January, 1982.

9. Herlihy, Maurice, and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems* (October 1982).

10. Lampson, Butler and H. Sturgis. Crash Recovery in a Distributed Data Storage System. Xerox PARC, April 1979.

11. Liskov, Barbara *et al. Lecture Notes in Computer Science. Vol. 114: CLU Reference Manual.* Springer-Verlag, Berlin, 1981.

12. Liskov, Barbara and R. Scheifler. Guardians and Actions: Linguistic Support for Robust Distributed Programs. Proceedings of Principles of Programming Languages Conference, ACM-SIGPLAN, 1982.

13. Parnas, D.L. On the Criteria for Decomposing Systems into Modules. *Communciations ACM* (December 1972).

14. Schell, Roger. Dynamic Reconfiguration in a Modular Computer System. Tech. Rep. TR-86, Mit Laboratory for Computer Science, 1971.

15. Stark, Eugene. Foundations of a Theory of Specification for Distributed Systems. Ph.D. Thesis, M.I.T. Laboratory for Computer Science, forthcoming

16. Weihl, William and B. Liskov. Specification and Implementation of Resilient, Atomic Data Types. M.I.T. Laboratory for Computer Science, Cambridge, MA., forthcoming.

17. Wulf, W. A., and M. Shaw. Abstraction and Verification in ALPHARD: Introduction to Language and Methodology. Carnegie Mellon University and USC Information Sciences Institute, 1976.