RECONFIGURATION IN ARGUS

Toby Bloom and Mark Day

1. Abstract

Early work on reconfiguration of distributed applications was done as part of the Argus project at MIT. We review that work, examine the lessons that we learned, and relate our Argus experience to current plans for building Thor: a distributed, highly-available object database management system. We emphasize the type system, communication model, and persistence model of Argus, and the interaction of those models with reconfiguration requirements.

2. Introduction

Argus is a language for building reliable distributed applications [1]. After a reliable distributed application has been built, it is useful to be able to replace parts of it while the application continues to provide service to clients. Such replacement and rearrangement is called dynamic reconfiguration [2]. Argus was not designed specifically to support dynamic reconfiguration; however, many of the constructs in Argus that support reliable distributed computing are also useful for dynamic reconfiguration.

In this paper, we review the work done on reconfiguration in Argus. On the theoretical side, we developed a set of correctness conditions for reconfiguration. On the practical side, we designed a full reconfiguration service, and implemented a more limited system. Our experience is relevant to the reconfiguration of other distributed systems with shared persistent data and statically typed module interfaces. In particular, it is relevant to Thor, an object database that shares these characteristics and is currently being designed at MIT [3].

The remainder of the paper is organized as follows: in section 3, we provide background information about Argus. In the following section, we explain what we mean by reconfiguration in Argus. In section 5, we informally present the theory that we developed for understanding the legality and effect of reconfigurations. In section 6, we describe the two reconfiguration systems that we designed, one of which we implemented; we compare the two systems in section 7. In section 8 we focus on the problems caused by the Argus type system. Section 9 describes related work, and we conclude in section 10 by describing some of the future problems we expect to encounter with Thor.

3. Background

An Argus program consists of a collection of *guardians*. A guardian is an abstract computational node, encapsulating some resource or data. Each guardian resides at a single physical node of a distributed system, and a single node may host multiple guardians. A guardian's interface to other guardians is a collection of remotely invocable procedures, called *handlers*. The language is statically typed, and the type

---

Toby Bloom (tobybloom@lotus.com) is with Lotus Development Corporation, 55 Cambridge Parkway, Cambridge, MA 02142, U.S.A.

Mark Day (mday@lcs.mit.edu) is with the MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, U.S.A.

system is essentially identical to that of CLU [4]. In Argus, as in CLU, there is no subtyping[1] .

Guardians are created dynamically by other guardians, and can destroy themselves. An incoming handler call at a guardian causes the guardian to fork a new lightweight process (thread) to service that call. There is a *catalog* service that maps string names and object types to objects, allowing a client to locate a server. In addition, handlers and guardians[2] are transmissible among guardians, so that a guardian can pass to another guardian its own handlers or the handlers of some third guardian. Note that there is no firm distinction between clients and servers: any guardian can act as a server for some other guardians, while being a client of still other guardians.

Guardians are expected to run indefinitely once started (as long as they do not deliberately destroy themelves) and must restart automatically when a node recovers from a crash. Some of a guardian's state is designated as "stable" and must remain consistent despite node and media failures. Argus uses a nested transaction system [5] to maintain consistency in the presence of distributed system failures.

4. Reconfiguration Requirements in Argus

Since Argus has the properties described in the previous section, the problems of reconfiguration in Argus are somewhat different from other languages and systems, which typically have less dynamic interactions among modules, weaker typing of interfaces, or less stringent notions of consistency in the persistent state. Creating and destroying program modules are already part of the programming model, and need no special support. Similarly, dynamic interactions ("connections") among modules need no special support. For example, the evolving philosophers problem [6] can be solved in Argus without needing any of the mechanisms we discuss in this paper, and thus does not fall into the class of problems that we refer to as "reconfigurations".

Nevertheless, replacing modules of a long-running application must preserve any persistent state, and should be accomplished with minimal disruption to the service provided by the application. Henceforth, we emphasize this problem of module replacement while preserving state.

We briefly summarize the requirements for correct replacement, and elaborate on them subsequently:

i) Clients of the old module must be able to interact with the new module transparently.

ii) Existing persistent state must be transferred from the old module to the new module. This may require reformatting the state.

iii) Type safety of interfaces must be preserved, because clients depend on the static specification of the services that they use.

iv) Groups of guardians must be replaced simultaneously, with their handlers being remapped onto new guardians in a nontrivial way.

---

[1] Argus does have a weak notion of *type inclusion* that covers certain cases where signatures of routines differ only in the exceptions that they signal.

[2] Strictly speaking, neither handlers nor guardians are transmitted: at least not in the sense of moving code or migrating the guardian. Instead, the name of the handler or guardian is transmitted. The effect is still that the client holding a handler or guardian object is able to call operations via that object.

The last requirement probably requires the most explanation: A guardian is the smallest unit of replacement that seems sensible in Argus, but it is not the largest unit of replacement. An application can be organized so that a group of guardians is cooperating to provide a single abstract service to clients. Such a group is called a *subsystem*; subsystems exist as abstract structures only, and are not directly represented in the Argus language. In the most general case, a collection of guardians may be replaced by a different number of guardians, some or all of which are at entirely different nodes. Further, the handlers in the new system may be grouped into guardians in a way that is entirely different from the old system; the only constraint is that every old handler that could be used by a client must have some corresponding new handler. We will consider in more detail what "corresponding" means, but intuitively it means "able to provide the same service": we would expect the new handler to have the same type as the old handler, and a specification similar to that of the old handler.

5. Theory

We feel that the literature on reconfiguration pays far too little attention to the semantics of changes made to a running system, questions such as "when is a change valid?" and "what is the impact of the change?" Before undertaking the design of a replacement service for Argus, Bloom studied the constraints imposed on dynamic replacement by the need to ensure behavioral consistency across replacements. In this section, we summarize the results of that study. We start by defining terms, then present an example, and discuss informal correctness conditions for replacement.

5.1 Terminology

We follow [7] in using the term *module* to describe a continuously-existing entity as seen by clients, while an *instance* is the collection of code and data that is replaceable. Over the course of its lifetime, a module is implemented by a series of instances. For every module in a correct replacement, the new instance replaces the old instance without affecting the state of the corresponding module. An instance's code is also referred to as that instance's *implementation*: this is in contrast to the corresponding module's *abstraction*, which is the specification of the module's type.

Thus:

A module's specification is an abstraction;
an abstraction is implemented by one of a number of implementations;
an instance consists of an implementation and some state; and
a module's runtime behavior is implemented by a series of instances.

5.2 An Example: UID generator

We consider a toy example to demonstrate the problems that can arise in determining correctness of replacement. Suppose that we have a unique-identifier (uid) generator, whose specification states that it will produce N distinct integers before repeating any result. An instance based on an implementation that starts at 1 and counts to N cannot replace an instance based on an implementation that starts at N and counts backward to 1, even though both are correct implementations of the abstraction[3]. For example, one generator may have produced 1, 2, 3 as uids. When the replacement occurs, the new instance must not produce 1, 2, or 3 as uids. However, the new instance need produce only N-3 distinct integers. So we can see from this that the specification to be met by

[3] The replacement is not possible in general, but these two instances can replace each other at two points in time: first, when no id's have been generated; second, when all N id's have been generated. These are rather trivial, though.

the replacing instance can be both stronger and weaker than the specification of the original abstraction. This new specification defines the *continuation abstraction*[4].

Now consider a situation in which two implementations have different specifications but can replace each other. For example, an instance of a module that produces only 10 distinct integers can replace an instance of a module that guarantees to produce N distinct integers, as long as the instance being replaced has already produced (N - 10) integers and the 10 integers produced by the replacement have not been produced already.

5.3 Correctness

We discuss the conditions for a correct reconfiguration informally; [7] presents a formal definition.

A simple, plausible, but too-limited definition of correct replacement is:

All abstractions remain unchanged across replacements

This definition means that the only legal replacements are changes of implementation, essentially swapping code and transforming state accordingly but leaving the abstraction unaffected. This definition is too restrictive in some cases and is insufficiently restrictive in others, as the uid example illustrates.

The next two sections describe the two situations in which we would like to relax this restriction, and modify the correctness condition accordingly. The two situations not covered by the above correctness condition are:

i) An implementation may meet the requirements for future behavior without correctly implementing the entire abstraction.

ii) An abstraction can be extended with compatible operations; clients using only the old operations are not affected

5.4 Future-Only Implementations

We want to allow replacement by an abstraction that is designed specifically to replace existing instances, and that will produce acceptable behavior when starting from some non-initial state. Since a correct reconfiguration requires only that *future* behavior be correct, we need not implement past behavior; so an implementation can be a correct replacement even if it could not have produced the module's past behavior. An implementation may satisfy the continuation abstraction without satisfying the original abstraction, and *vice versa*, as shown in the uid example. However, any implementation always satisfies its own continuation abstraction at all times, since it can be replaced by an exact copy of itself.

5.5 Invisible Extensions

We would also like to be able to extend abstractions, adding new operations. As long as such extension logically does not affect the existing clients of the module, we would like to be able to perform such a replacement as though we were only changing the

---

[4] Note that "continuation" in this context has nothing to do with the flow of control in a program, and is unrelated to continuations as used in denotational semantics or programming languages such as Scheme.

implementation of an abstraction. There are two ways that a new operation of a module can be visible to an existing client:

i) by changing the state of the module to a value that could not have existed previously, or

ii) by altering the type of an object returned by an existing operation.

An example of the first kind of visibility is a new operation that deletes entries from a table. Adding such an operation means that a lookup of key *k* can fail even though *k* has previously been added. Clients that rely on the table always increasing in size may break when the new operation is added. However, these problems can only arise if the new operation is actually called.

It is tempting to assert that an operation is visible if it alters state, and is invisible otherwise; but this is incorrect. An abstraction may specify that a client cannot deduce certain kinds of information (the next id from the unique-id generator, for example) and the addition of an operation disclosing that information is a visible change, even though it does not alter any state. Correspondingly, operations that simply store to and fetch from a new field of an object are altering the state of the object, but those operations and the new field are not visible to other clients.

The second kind of visibility is somewhat more difficult to explain. Since Argus does not have subtyping, the addition of a new operation to a module produces a new module type that is not compatible with the old type. If some existing operation (of the module being replaced or another module) accepts or returns objects of the module type, the type of the new object provided to/by the client will not match the type of the object expected by the implementation. This means that the change is visible even if no new operation is ever called.

We have now presented the problems with our first correctness condition. Our actual correctness condition is:

Continuation abstractions are preserved or invisibly extended by a replacement.

5.6 Discussion

In this section we consider general issues of reconfiguration and its relationship to correctness, emphasizing that much of our understanding of replacement contradicts or limits our original intuitions.

Some Intuitively-Correct Replacements Violate Typing: Even with our extended definition of correctness, many logically safe replacements will violate static typing. For example, adding a handler to a guardian is not type safe if the guardian is ever passed as an argument (recall that there is no subtyping in Argus).

Specifications Are Related to Initial States: The notions of abstraction and specification change when continuation abstractions are introduced. In the absence of replacement, specifications implicitly assume that instances start in an initial state defined by the creation operation(s). In the presence of replacement, the instance starts with some state defined in part by another instance and in part by the reconfiguration process. For example, replacement of a maildrop guardian should preserve the state of all the mailboxes stored there; we do not want the replacement maildrop to be created empty, even though that is the normal state of a newly-created maildrop.

History Rules Out Some Intuitively-Correct Replacements: As a result of working with the formal model of [7], it became apparent that there are two potential problems with

replacement from a strictly semantic point of view. First, it may be impossible to replace one implementation with another, even though both are correct implementations of the abstraction. Second, the future behavior of a replacing instance is constrained by the past behavior of all replaced instances; unless a history of replacements is kept, it may be impossible to ensure the correctness of a replacement.

When is it impossible to replace one implementation with another? This can occur because the replacing implementation may not have any starting state that corresponds to the ending state of the replaced implementation. For example, a unique-id generator that uses a random-number generator and a table of generated numbers can produce an arbitrary sequence of uids. If we want to replace such an implementation with an implementation that counts from 1 to N (perhaps because we've noticed the large amount of storage being consumed by the table in the first implementation), we find that it is impossible. We know all of the numbers produced, but we cannot derive a starting state (a value for the internal counter) from that information; the generator is in an abstract state that the new implementation cannot reach. An alternate source of the same problem is that a single concrete state of the implementation being replaced may correspond to a set of abstract states, but those abstract states are implemented by two or more concrete states in the replacing implementation. In this situation, we cannot decide at replacement time which abstract state the module is in, and so cannot produce a unique legal starting state.

If there have been multiple replacements, we may have difficulty ensuring correct future behavior. For example, a unique-id generator has counted from 1 to 12, and has been replaced by an implementation counting from N downward. Such a replacement is legal, since the specification requires only that N distinct values be produced. When the new instance is itself replaced, its code and state reveal only that it counts down from N, and has reached the value N-3. Replacing it with the original implementation and starting again at 1 appears to be legal, but will be incorrect. Only by knowing the entire history of replacements can we avoid this problem.

6. Practice

There have been two different efforts to provide support for reconfiguration in Argus, taking different approaches to the problem.

6.1 System-Supported Replacement

First, Bloom designed a complete replacement service that would replace any active guardian or subsystem, subject to certain type correctness conditions [7]. We refer to Bloom's approach as System-Supported Replacement (SSR). With SSR, all guardians are replaceable. There are two components to SSR: special hooks in the Argus system that allow replacement, and an interactive facility for the user to coordinate the replacement.

The system hooks are not complex, but they have a cost. Handlers must have a level of indirection so that new handlers can be associated with existing handler identifiers (hids) dynamically. A straightforward implementation incurs an extra cost on every handler call. Bloom proposed that handlers be called directly via the hid, with old hids invalidated so that they cause a fault when invoked. The system would then reroute the call without affecting the caller, and send a message to the calling system so as to update the hid. This approach increases the cost of handler calls immediately after a replacement, but has no effect on normal operation.

The interactive replacement facility allows the user to reconfigure the system. The user selects the guardians and subsystems to be replaced and quiesces them. Quiescing can mean forcing the abort of all active transactions, or it can mean waiting for the active

transactions to complete. Note that in Argus, all clients must be prepared for an abort because communication failures or machine crashes can also abort transactions. After quiescing the guardians to be replaced, the user creates the new (replacing) guardians, and connects handlers from the old guardians to handlers in the new guardians. The types of old and new handlers must match, but the set of handlers from one old guardian can be distributed among several new ones, and *vice versa*. The user then transfers state from the old guardians to the new guardians; this may involve providing procedures to compute new state from old state. Finally, the user activates the new set of guardians. Client computations proceed as if the services they were using had been temporarily unavailable, and had just become available again.

6.2 Application-Level Replacement

At the time Bloom's work was completed, a working prototype of Argus already existed and did not provide the necessary low-level support required by Bloom's design. So Day conducted an experiment [8] to determine whether it was possible to build reconfigurable applications without low-level support. We refer to Day's approach as Application-Level Replacement (ALR). ALR requires that implementors of replaceable guardians provide special operations. The extra operations extract the abstract state from the guardian and create a new guardian from this abstract state[5]. This amounted to using the ideas invented for transmitting values of abstract types [9] to transmit the persistent state of a guardian from the present instance to the future instance. The rebinding of handlers occurred as clients tried to use handlers of a replaced guardian and received exceptions: a client's first fallback was to look in the Argus catalog to try to find the service again. This is a typical strategy for a replicated service, but is less commonly used for reconfiguration.

Because the Argus implementation allows the replacement of code in a crashed guardian, the operation to extract the state object typically is unimplemented until the time of replacement: the guardian to be replaced is crashed, the normal code replaced with a replacement implementation, and the guardian restarted[6]. Then the extracting handler is called, returning the state of the guardian in some convenient form. To allow for flexibility in implementation, the type of the state object is the Argus type meaning "any transmissible object" in the signatures of both the state-extracting and guardian-creating operations. That is, we sacrifice static type-checking at that interface since we have no way to predict the most convenient way to return the guardian's state.

ALR requires more participation by both guardian designers and clients of guardians. However, the requirements are not drastically greater than for conventional, non-reconfigurable, Argus applications. Type designers are already expected to provide a number of "standard" operations, such as copy, equal, and print, so introducing a "standard model" of guardian construction is not a significant loss of freedom. Similarly, clients of a guardian must already be prepared to deal with the exceptions *failure* (permanent error) and *unavailable* (temporary problem) on every handler call; ALR takes advantage of that structure to accomplish rebinding without system support, on demand as the handlers are called.

---

[5] In addition, existing low-level support in the Argus prototype was used to crash guardians, restart guardians, destroy guardians, and alter the code associated with a guardian (this last operation can be done only while a guardian is crashed).

[6] Typically, the replacement implementation also contains different implementations for the normal operations of the guardian, so that they signal "unavailable" immediately. This effectively implements Bloom's mechanism of taking a guardian out of service, though less elegantly.

7. Comparison

ALR can be viewed as a lower bound: it represents what can be achieved in the existing Argus prototype and probably in any plausible implementation of the language. SSR is more of an upper bound, and probably represents the limits of what can be done within the language definition (notably its type system).

ALR requires more from reconfigurable modules, but less support from the system. Its major advantages are:

i) There are no special requirements on the Argus implementation, which can therefore be smaller and simpler.

ii) There is no performance overhead in normal operation. A system that is not reconfigurable or that is never reconfigured pays no cost for the reconfiguration of other systems.

The disadvantages are:

i) The approach only works for guardian replacement: there is no good way to deal with the replacement of a subsystem consisting of several guardians.

ii) Rebinding of handlers depends on clients handling exceptions appropriately. Robust clients must handle such exceptions anyway, but more casual clients will die with an unhandled exception in situations where a more elegant approach, hiding the handler rebinding entirely, would allow them to continue executing.

SSR requires less from reconfigurable modules, but more from the rest of the system. Its major advantage is:

i) Any module can be replaced, so reconfiguration is more generally available and need not be anticipated by the developer.

Its disadvantages are:

i) Handlers must be implemented to be redirectable. This implies a level of indirection that adds to the complexity of the Argus implementation and can reduce its performance.

ii) Forwarding information must be maintained in spite of inaccessible nodes or partitioned networks, probably requiring some form of highly-available location service [10].

Both methods have the drawback that potentially complex procedures need to be written at replacement time, if the persistent state is being restructured. The ALR approach can sometimes avoid this by defining, when the guardian is written, a canonical external representation for the guardian's abstract state. A replacement that does not affect the canonical form is then simpler.

8. Type Issues in Argus

In this section, we consider the areas in which Argus's type system and fundamental design decisions limited flexibility for reconfiguration. We re-emphasize that Argus was not originally designed to support reconfiguration, so that these limitations are not surprising. However, we would like to understand how we might have designed Argus differently had we planned for more dynamic reconfiguration.

8.1 Would subtyping have helped?

When considering the problem of extending an abstraction, it appears to be exactly the sort of problem that is handled by subtyping. The extended abstraction is a subtype of the original abstraction, and can be used everywhere that the original abstraction was used. Subtyping of guardian interfaces would have made reconfiguration mechanically easier, since some part of the system would be ensuring the correct visibility of handlers in interfaces. However, the same semantic issues discussed in section 5.5 would arise: the new handlers must not betray their existence to old clients, and subtyping does not affect this.

Subtyping would also have helped to eliminate certain kinds of recompilation mandated by changes in argument and return types. Any changes to replace argument types with subtypes, or to replace return types with supertypes, could have been done transparently. Experience with ALR suggested that this difficulty rarely arose. Since handlers communicate abstract values, the actual implementations of arguments can be altered and refined on either side of the handler call, without affecting the interface in most cases. Most of our reconfiguration problems arose from changes to the guardian interface, changes in the number of arguments or results, or changes to types where the old and new type did not have a straightforward subtype relationship.

8.2 Would default arguments have helped?

If Argus handlers had notions of optional arguments, default arguments, or "rest" arguments, then even more reconfigurations would have been possible, although the guarantees provided for interfaces would have been weaker. In a distributed system, where a call is expensive and consumes resources on more than one machine, you'd like to be sure that your calls make sense, and default arguments can lead to incorrect assumptions. This is an example of how static typing of interfaces is a way to increase reliability in a distributed system, but it is in conflict with allowing easy reconfiguration.

8.3 Would sets of handlers instead of guardians have helped?

The decision that had the greatest (negative) impact on reconfiguration was emphasizing guardian objects as a mechanism for grouping and using handlers, then using strict CLU-style typing (with no subtyping) for deciding the compatibility of two guardian objects. This means that every change to the signature of any handler -- whether it was the types of arguments, the number of arguments, or adding new handlers -- produced a new and incompatible type, forcing a change to be visible to clients that were otherwise unaffected by the change (typically because they didn't use the changed handlers).

As stated earlier, subtyping would have mitigated some of these problems. An alternate solution is to explicitly construct collections of handlers as needed, rather than using guardian objects. We experimented briefly with an interface to the Argus catalog that allowed a client to look up a service and receive a token instead of a guardian object, then use that token (corresponding to a particular guardian) to look up specific handlers of that guardian. This worked but was extremely complex, since each catalog operation is parameterized by the type of the object being requested. This is similar to an application-level implementation of subtyping; but in contrast to application-level reconfiguration, we found this intolerable without language support of some kind. The advantage of supporting collections of handlers instead of subtyping is that there are fewer semantic issues to be resolved; however, such collections are *ad hoc*, less flexible, and less powerful than subtyping.

8.4 Would dynamic dispatching have helped?

Dynamic dispatching of calls (only checking the arguments to a handler at call time) would make reconfiguration simpler, but at the expense of causing more problems at run time. As with default arguments, there are good reasons for working to ensure that a call makes sense before sending it to waste another machine's resources.

9. Related Work

CONIC [2, 6] is a system that splits the Argus computation model into two parts: a relatively static model of modules and connections for normal processing, and a separate facility for rearranging those modules and connections into a different configuration.

We believe that the key advantage of CONIC compared to Argus is that separate ports are genuinely separate, whereas our handlers are usually glued together into guardian objects where changes in one affect the others. While we believe that CONIC works well for its application domains, its relatively static view of computation and communication would be awkward for expressing our applications. We are not convinced that CSP-like computation models are the *only* appropriate models for concurrent and distributed systems, so that reconfiguration *must* be expressed in a CSP-like fashion [11].

We are also unsure, for our applications, whether it is a good idea to separate the computation language from the reconfiguration language. Most of our reconfigurations have been performed by hand, but it seems that a reconfiguration program could be expressed in Argus with no particular difficulty. A normal computation in Argus can deal with creating guardians on particular nodes, locating objects, and other activities that seem to be present only in the reconfiguration language in CONIC.

In the Mercury system [12], all communication is done via dynamically-created ports. The system model is thus rather similar to Argus if we eliminate guardian objects altogether, and weaken the typing of handler interfaces. A problem arises that we can illustrate with the following example: there may be many instances of a container service, and all of them may register individual ports called store and fetch in the catalog. A client doesn't care which container it uses, but the store and fetch must be associated with the same container. This requires either registering ports for containers (when called, such a port returns the associated store and fetch, so that a client never asks for a store port or a fetch port) or it requires manifesting the relationship between ports in their names, so that a client asks for "fetch22" and "store22". So Mercury does not entirely solve our reconfiguration problems.

Recent work by Purtilo and Hofmeister [13] has also used Herlihy's model of abstract value transmission [9] to allow the transmission of processes. Many of the problems that complicate their work are finessed by the Argus transaction system, which provides a clean and simple notion of both quiescence and state: when all else fails, we can deliberately crash a guardian and be ensured of a consistent state for restarting. However, Purtilo and Hofmeister's work is more widely applicable than ours, since it deals with programs written in C. We might view their work analogously to ALR, and call it the next stage of "building at the application level." In this analysis, Purtilo and Hofmeister are exploring how much reconfiguration can be done following Herlihy's model but without having the facilities of a language like Argus.

Much of the literature dealing with reconfiguration considers only how to rearrange modules and their connections, without considering persistent state in those modules. Some ideas that are similar to our limits on replacement are formulated in [14], where the notion of *upward compatibility* of software changes is formalized. However,

because those authors are considering only code and not persistent state, they do not identify the problems caused by the history of replacements. We conjecture that upward compatibility is a special case of our correctness condition.

10. Ongoing work

Our ongoing work takes place in the context of Thor, a project to build a distributed object store that will allow clients written in different languages to share objects. We are currently examining two new areas.

First, we need to understand how reconfiguration applies to applications using a persistent store. We plan to have a type hierarchy in our language for defining objects, and we hope that it will solve some of the problems that we had with Argus; but we also have new problems, since persistent objects of the same type may have different implementations. This problem also arose in Argus, but objects of the same type could have different implementations only if they were in different guardians; within a guardian, a single abstract type has a single implementation, and objects of the same type in other guardians are inaccessible (only abstract values of the type can be transmitted between guardians). A similar single-implementation-of-type container construct may be worth introducing into Thor so that we can eliminate dynamic checking of implementations within that construct.

Second, our previous work on moving persistent state assumed small enough quantities of data so that all translation and reformatting could be performed at reconfiguration time. Since we expect to handle large volumes of data with Thor, we need to consider how to handle this task when there is too much data to handle all at once. We will probably want to translate data into new formats lazily [15], meaning that several replacements might take place between accesses to an object and many implementations of a type will exist in the system at the same time.

11. Acknowledgements

This paper is based on theses supervised by Barbara Liskov. We thank Sanjay Ghemawat, Deborah Hwang, Pierre Jouvelot, and Umesh Maheshwari for careful reading and helpful comments.

12. References

[1] Liskov, B. "Distributed Programming in Argus." *Communications of the ACM* 31, 3 (March 1988) 300-313.

[2] Kramer, J. and J. Magee, "Dynamic Configuration for Distributed Systems." *IEEE Transactions on Software Engineering* 11, 4 (April 1985) 424-436.

[3] Liskov, B., R. Gruber, P. Johnson, and L. Shrira, "A Highly Available Object Repository for Use in a Heterogeneous Distributed System." in A. Dearle, G.M. Shaw, and S.B. Zdonik (eds.), *Implementing Persistent Object Bases: Principles and Practice*. (Morgan Kaufmann, San Mateo, CA, 1990) 255-266.

[4] Liskov, B., A. Snyder, R.R. Atkinson, and J.C. Schaffert, "Abstraction Mechanisms in CLU." *Communications of the ACM* 20, 8 (August 1977) 564-576.

[5] Moss, J.E.B. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Mass., 1985.

[6] Kramer, J. and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management." *IEEE Transactions on Software Engineering* 16, 11 (November 1990) 1293-1306.

[7] Bloom, T. "Dynamic Module Replacement in a Distributed Programming System." MIT Laboratory for Computer Science TR-303, March 1983.

[8] Day, M.S. "Replication and Reconfiguration in a Distributed Mail Repository." MIT Laboratory for Computer Science TR-376, April 1987.

[9] Herlihy, M. and B. Liskov, "A Value Transmission Method for Abstract Data Types." *ACM Transactions on Programming Languages and Systems* 4,4 (October 1982) 527-551.

[10] Hwang, D.J. "Constructing a Highly-Available Location Service for a Distributed Environment." MIT Laboratory for Computer Science TR-410, January 1988.

[11] Jacob, J. "A Model of Reconfiguration in Communicating Sequential Processes." *Information Processing Letters* 35,1 (June 1990) 19-22.

[12] Liskov, B., T. Bloom, D. Gifford, R. Scheifler, W. Weihl, "Communication in the Mercury System." *Proceedings of the 21st Annual Hawaii Conference on System Sciences* (1988) 178-187.

[13] Purtilo, J.M. and C.R. Hofmeister, "Dynamic Reconfiguration of Distributed Programs." *Proceedings, 11th International Conference on Distributed Computing Systems*, Arlington, Texas (May 1991) 560-571.

[14] Narayanaswamy, K. and W. Scacchi, "Maintaining Configurations of Evolving Software Systems." *IEEE Transactions on Software Engineering* 13,3 (March 1987) 324-334.

[15] Skarra, A. and S. Zdonik, "The Management of Changing Types in an Object-Oriented Database System." *Proceedings of the Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA'86)*, Portland, Oregon (September 1986), 483-496.