Planning for change: a reconfiguration language for distributed systems

Distrib. Syst. Engng 1 (1994) 313–322. Printed in the UK

Planning for change: a reconfiguration language for distributed systems

B Agnew†, C Hofmeister‡ and J Purtilo†

† Computer Science Department and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA
‡ Siemens Corporate Research, 755 College Rd E., Princeton, NJ 08540-6632, USA

Abstract. To improve the programmer's use of reconfiguration methods in distributed systems, we are studying notations for expressing change in the form of *reconfiguration plans*. These plans are used to describe the application's reconfiguration as a result of the detection of events from either the application itself or its environment. Our current work on this subject is a system called Clipper. Clipper is based on C++, and provides the programmer with a language for describing reconfiguration plans that are compiled into the run time mechanisms for implementing change rules in the application. This paper describes Clipper and the requirements that guided its development.

1. Introduction

Dynamic reconfiguration of an executing distributed application entails mapping one application configuration to another, where an application configuration refers to the structural properties of the application. The mapping may be applied to implement a planned upgrade, to recover from a fault condition, or to adjust performance in the application environment. Given the advent of environments in which dynamic reconfiguration is possible, our interest has focused on methods for assisting the programmer in the control of reconfiguration. Specifically, we are concerned with the mapping of *reconfiguration plans* to the execution of the application. To facilitate this, we have developed a simple C++ [3] extension for organizing reconfigurations on behalf of programmers, based upon recognition of reconfiguration events. Reconfiguration events represent an application state, or precondition, that is necessary for transformation of the application. These events are bound explicitly by the programmer to a plan of actions appropriate for the transformation on the application structure.

We adopt the terminology from the prior reconfiguration research [6]. A distributed application consists of a set of processes (possibly across many heterogeneous host platforms) interconnected by communication channels. The processes share only the distributed environment in which they execute, and their mutual bindings. Dynamic reconfiguration may involve change to the structure of the application (such as addition and deletion of modules or bindings) or it may involve altering how the structure is mapped onto the underlying host resources.

In our previous work [12], we focused on the run-time environment required to support dynamic reconfiguration. This environment provided the programmer with a library of system-call-like accessors to effect change. A component of the application using these methods was referred to as a *catalyst* module. The purpose of the catalyst module in an application is to recognize conditions suitable for reconfiguration and perform the operations necessary to satisfy those conditions. Our experience has shown that construction of these modules requires much repetitious coding; this suggests that the task can be made more efficient by creating a higher level abstraction for translating the programmer's reconfiguration plans automatically. Clipper is our first experimental notation for expressing these reconfiguration plans abstractly.

Clipper represents an application configuration as a collection of C++ structures, and allows the programmer to describe mappings between configurations as operations on structures. Then each mapping is explicitly associated with the conditions and events that will trigger it, resulting in a reconfiguration plan. The plans are compiled into a catalyst module which is executed with its subject application. At run time the catalyst executes the plans after recognizing the events associated with those plans. With this method, the programmer is free to realize the transformation of the application at the configuration level without additional concern of implementation details.

This paper describes Clipper, along with requirements we have identified that led to its development. It describes how catalyst modules are constructed, and subsequently executed. It discusses the current state of our overall system and the experiments we are performing.

0967-1846/94/050313+10$19.50

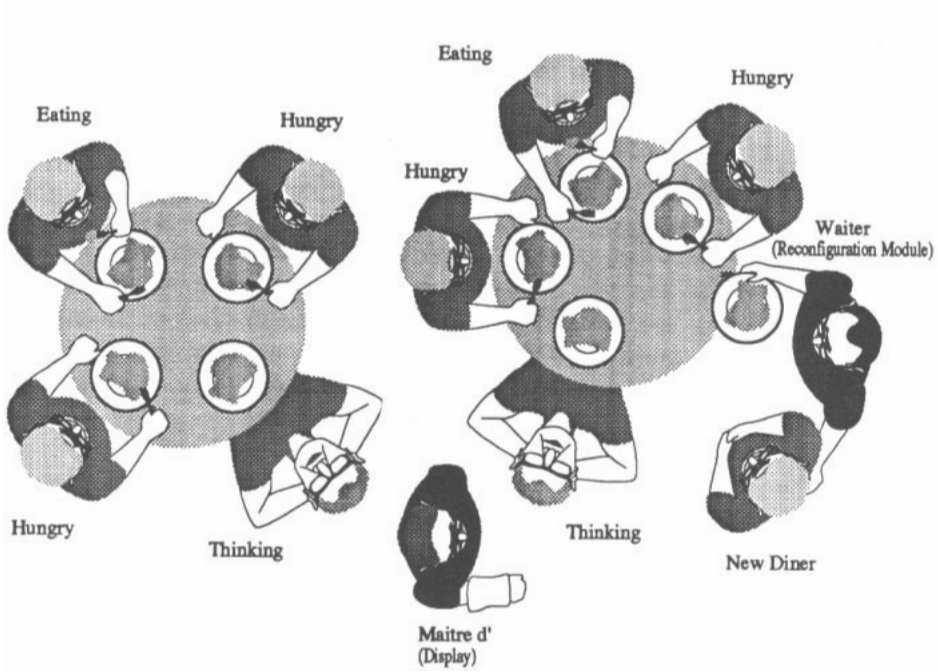

Figure 1. The dining philosopher's initial configuration (left); during reconfiguration (right).

2. Requirements of the reconfiguration language

The requirements we chose for the Clipper language were motivated by these considerations: developmental ease, protection of the application state, clean separation between (normal) application processing and reconfiguration activities, and minimization of side effects due to reconfiguration. As with any language development effort, we sought a notation that is both expressive and succinct: a language detailed enough to support all structural changes to the application but simple enough that syntactical errors can be kept at a minimum.

We also believed it advantageous for the language's syntax, scope and control structures to be similar to high-level languages currently in use; this permits the programmer to develop reconfiguration plans with some intuition as to their behaviour. All changes in the application are orchestrated from a central plan external to the application, differing from the notations expressed in Durra [1] and Gerel [4]. This simplifies the set of reconfiguration commands [5] and enables portability between distributed environments.

Of course, not all types of reconfigurations are appropriate or correct for all applications, and therefore an important requirement of the language and system is to protect against inappropriate operations. Constraints on the set of reconfigurations valid at any given point can be described as arising from two sources. In the first, the application is assured of *semantic consistency*, independent of the nature of the application. For example, a module cannot be connected to another module that does not exist. In the second form of protection, the application configuration is limited to the set of possible configurations specified by the reconfiguration plan. This protection is explicitly enforced by the programmer, and is application dependent.

Another general requirement is one of non-interference: the steady state behaviour of a reconfigurable application should not vary significantly from a similarly structured static application. Specifically, during the normal operation of the application users should not experience performance or reliability degradation due to the presence of any external system in the run time environment that would make the application reconfigurable.

The reconfiguration process itself should be efficient in resource and time use. This is not a strict requirement, and is intended to reflect a strong sentiment that most reconfigurations (other than those intended to alter functionality) should be invisible to users of the application.

We use a dining philosophers example to illustrate the intended application of a reconfiguration language. In particular, it illustrates how use of such a language would augment the current run-time environments that support dynamic reconfiguration of distributed applications. (In the past, this example has been referred to as the 'uninvited diner' problem.) The initial application consists of four components representing dining philosophers (figure 1, left). Each diner module has three possible states: eating, thinking, or hungry. Adjacent diners share a resource, a fork, and each diner must have exclusive use of its two forks in order to eat. After eating, a diner reverts to a thinking state, where it remains until it becomes hungry and requests the two forks.

As a reconfiguration operation, we would like to add a new diner to the application (figure 1, right). To support this change, the following modifications to the application are necessary: replacing the original diner

module with one that it is capable of transferring its state (which is needed to maintain the correct sharing of forks), adding a display (Maitre d') module to request reconfiguration information (such as the name of a new diner) and adding a catalyst module (Waiter) to perform application configuration changes.

We have modified the original module description of the diner to allow distinction between a dynamically created object and a statically allocated one. This is important for synchronization of the new diner with the existing application. A new diner module checks its status at the start of its execution. If status is dynamic, then the module waits for state information from the diners to its left and right before proceeding. This is necessary to prevent the application from entering a state of starvation. The new module enters one of its three working states once it receives state information (a token indicating a fork) from its neighbouring diners. The reconfiguration operation is described by the following steps.

(i) Get access to a new diner and the adjacent existing diners. Change an attribute of the new diner to indicate that the diner is dynamic.

(ii) Activate queued binding mode. This allows us to defer binding changes until we are ready to add them just before starting the new diner.

(iii) Disconnect the two adjacent diners. Connect the new diner to the existing diners.

(iv) Release queued binding mode. This applies all previous binding changes since the activation of the queued binding mode.

(v) Add the new diner.

(vi) Synchronize the new diner with the application by inducing the existing diners to dump their state information to the new diner.

This type of reconfiguration was enabled by prior work, such as found within the Conic system [8] or the Surgeon system [6]. In the terminology of our own platform, the catalyst for this system would need to be prepared manually; more importantly, the programmer must decide the basis for initiating the introduction of a new diner. This is not an insurmountable problem, and indeed we have solved it in the past via manual techniques. Nevertheless, the problem still remains: even though the programmer can reason about the reconfiguration in the abstract, the only medium for expressing decisions on reconfiguration was the programming language used to implement the application itself.

The programmer should be able to simply declare the names of abstract *events*, then illustrate in terms of the structure how the application should change in the face of those events. The programmer should be able to express reconfiguration plans abstractly, and then have implementation of the appropriate infrastructure installed automatically in the context of the particular application. The problem of identifying the run time mechanisms needed to recognize events and trigger reconfiguration is then solved separately.

By separating the development of the application components from the task of enabling application reconfiguration, we are exploiting a 'divide and conquer' approach to development. In our example, one event might be called 'arrival' (of some new diner); later, the programmer may wish to provide rules for other events, such as recognition of a fault (e.g., 'host down', mapping into a restart operation for whatever processes are affected) or load balancing (e.g., 'loaded' mapping into a process migration plan). Ideally, any language for planning reconfigurations should support expression of rules such as these, and allow for the executables to be prepared without manual intervention by the programmer.

3. Design and implementation

Clipper is the language and system intended to meet requirements listed in the previous section. Our current notation is a prototype, and is based upon a simple extension to C++; our language for the configuration itself, as well as the run time environment to support the reconfiguration steps, is drawn from our work in Polylith [12]. This section of the paper describes the language itself, the event mechanism, and the properties of catalyst modules that are generated by the Clipper system for introduction to the application run time environment.

3.1. Description of the reconfiguration language

To meet the requirements stated earlier, C++ was chosen to support the reconfiguration language. C++ provides type checking, restricted data access, overloading of operators and functions, derivation of data types, and implicit initialization of data structures. Type checking permits rudimentary static analysis of the reconfiguration plan, while data access, operator and function overloading allow implementation of dynamic analysis. Derivation of data types helps enforce consistency in the definitions of application components. Consistency is needed to ensure all application components can be analysed dynamically. Implicit initialization of data structures lets the user take advantage of default values specified in the configuration file.

By using an existing language (that includes an overloading capability) we were able to construct the first prototype of the system rapidly. Then, based upon our experimentation with Clipper, we slightly modified the syntax into a rule-based notation more strongly suited to the expression of plans. We use the diners application to illustrate the nature of Clipper. In table 1 the code on the left side is explained by comments on the right. We have numbered the comments to indicate the steps involved in the reconfiguration.

A plan is a valid C++ program in which changes to application components (modules and bindings) occur as a result of manipulations on a set of class objects representing the application. The set is the *model* of the application, and it is through this model that all changes to the application are made. The Clipper compiler

Table 1. The reconfiguration plan for the dining philosophers example.

The reconfiguration code	The code behaviour
#include "dinnerplans.h"	include model object definitions
void add_diner	
(char *left_name, char* center_name, char* right_name)	
{	
diner oldleft(left_name);	1 get access to diner objects
diner oldright(right_name);	1
diner newdiner(center_name);	1
*newdiner.STATUS= "dynamic";	1 set attribute STATUS
init_bind();	2 lock bindings
oldleft.unbilink("right_req" "left_req", oldright);	3 disconnect oldleft from oldright
oldright.unbilink("left_fork", "right_fork", oldleft);	3 disconnect oldright from oldleft
oldleft.bilink("right_req", "left_req", newdiner);	4 connect oldleft to newdiner
newdiner.bilink("left_fork", "right_fork", oldleft);	4 connect newdiner to oldleft
oldright.bilink("left_fork", "right_fork", newdiner);	4 connect oldright to newdiner
newdiner.bilink("right_req", "left_req", oldright);	4 connect newdiner to oldright
set_bind();	5 apply binding changes
newdiner.start();	6 start new application component
oldleft.dumpstate("reconfig_right", newdiner);	7 dump application state
oldright.dumpstate("reconfig_left", newdiner);	7 dump application state
}	

builds the class definitions and model objects when it parses the static representation of the application. Each model object represents an executable component of the application. A model object type is a class containing attribute, and interface objects. These internal objects are initialized according to the appropriate module description given in the static representation.

We now use the diner reconfiguration plan to illustrate a typical reconfiguration sequence in a Clipper plan. The first step of the plan involves obtaining access to the model elements representing the parts of the application we wish to change. This is done by declaring three diner objects, each initialized through a constructor. The character string passed to the constructor identifies the module that this diner object represents. If the module is currently running, or is described by an existing model object the application model, this diner object will alias it. If the module does not exist (in the distributed environment or model), this diner object will represent a new module in the application. The scope of model object types extends beyond that of normal static types. The distributed object's lifetime dictates the lifetime of the model object representing it, although local identifiers aliasing that model object exist only in the block in which they are declared.

After having gained access to the application, we perform changes in the remaining steps. We queue the subsequent binding changes in step two to prevent the application from reaching unwanted intermediate configurations. These binding changes are then applied in step five, just before starting the new diner. In steps three and four the existing diners are disconnected from each other and reconnected to the new diner. After the new diner is started in step six, we command the two older diners to disgorge their state information to the new diner to maintain the consistency of the application.

Table 2 illustrates the commands for manipulating module objects. The base_module class is the base class from which all modules described in the static application are derived. By using a base class for all model changes we were able to better control the consistency of the application. All operations on the components of the application and their member attributes and interfaces are atomic in the reconfiguration environment to prevent them from conflicting with each other.

3.2. Events initiating reconfiguration

There are three possible sources of stimulus for reconfiguration: the application, the distributed environment, and the system environment. Events received by the catalyst are acted upon immediately unless a reconfiguration operation has already been initiated. If this is the case, the events are queued in the order of their arrival until the current reconfiguration is complete.

In Clipper, reconfiguration plans execute by implicit invocation. We bind the events that initiate reconfiguration with the reconfiguration plans using *bind statements*. Table 3 shows the text for binding an event called "arrival" to the reconfiguration plan **add_diner**. We first declare an event object with the internal name of "arrival". Then, we bind that event object to a reconfiguration plan by creating a binding object. Binding object construction requires an event object and a reconfiguration plan. If we need to send any of the event's parameters to the plan we also include a tape code (a string specifying parameter types in the distributed system) to specify the format of those parameters. For our example, the tape code is 'SSS', which represents the names of the two adjacent diners and the new diner.

The reconfiguration plan **add_diner** is called with the following sequence. The event generating module display receives a request from the user to add a new diner. The event generating module then broadcasts the event "arrival" throughout the distributed system. The

Table 2. The methods for modifying the application in Clipper. All objects representing application executables are derived from the class **base_module**.

The Function protoype and description
base_module::start(); Add this module to the application.
base_module::stop (); This module is removed from the distributed environment.
base_module::link (`char*` *from_iface* , `char*` *to_iface* , `base_module&` *to_module*); Add a directed binding between one of this module's interfaces and another module's interface.
base_module::unlink (`char*` *from_iface* , `char*` *to_iface* , `base_module&` *to_module*); Remove a directed binding between one of this module's interfaces and another module's interface.
base_module::bilink (`char*` *from_iface* , `char*` *to_iface* , `base_module&` *to_module*); Add a bidirected binding between one of this module's interfaces and another module's interface.
base_module::unbilink (`char*` *from_iface* , `char*` *to_iface* , `base_module&` *to_module*); Remove a bidirected binding between one of this module's interfaces and another module's interface.
base_module::dumpstate (`char*` *from_iface* , `char*` *to_iface* , `base_module&` *to_module*); Create a temporary binding between *from_iface* and *to_iface* from this module to *to_module* .
base_module::dumpstate (`char*` *from_to_iface* , `base_module&` *to_module*); Create a temporary binding between *from_to_iface* and *from_to_iface* from this module to *to_module* .
init_bindings (); Queue all binding changes applied after this function call.
set_bindings (); Apply all binding changes made after the previous call to init_bindings.

Table 3. The Bindings file for the dining philosophers example. We have bound the event "add" to the plan **add_diner.**

The binding code	The code behaviour
#include "dinnerplans.h"	declarations of plans for binding objects
R_event* e1= make_event("arrival");	creation of an event labelled add
make_binding(e1, "SSS", add_diner);	creation of a binding between the event e1 and the plan add_diner

parameters sent with the event are the name of a new diner and two adjacent diners. The catalyst module receives the event and executes the **add_diner** plan, which is the only plan bound to that event in the bindings file. Execution of a reconfiguration plan involves parsing the event parameters in the format specified by the binding's tape code and calling the binding's plan with the result. The catalyst resumes waiting for additional events after **add_diner** terminates.

In the case of our example, we have only one plan for all operations pertaining to the "arrival" event. But, in the case of more complicated reconfigurations, it may make more sense to break up the configuration changes into smaller, more reusable components. This is especially true if those components are needed for other events. For example, deleting a new diner requires a state transmission similar to adding a new diner. If we were to change the example to include removal of a diner, the changes required to the reconfiguration plan would be made simpler by separating plans in this way, especially if the state transmittal was complex.

The second source of events is the distributed environment. Events generated by the distributed environment are received in the same manner as application events, although in this case our system relies upon the underlying bus mechanism to recognize and generate the appropriate event stream. An example of a distributed environment event would be the sudden failure of a module (this could also be classified as a system event, since the fault is visible at the system level as well). Once it has detected the problem, the bus sends an event to the catalyst. The plan bound to this event is then activated. Typically, such a plan reconfigures the application based on the information passed to it in the event, such as the type of module that had terminated and any special termination conditions.

The third source of events is the system environment. A system environment event is a trigger representing a change to the operating environment of the modules making up the application. An example of such an event would be the failure of a module's host machine. We detect this by executing a separate module with the application, one that is capable of generating system status events. Determining a set of realistic events is a complex task for the the underlying system. Currently, the reconfigurable system on which Clipper is implemented recognizes only a limited set of events; a more practical implementation would need to include a more robust fault tolerance and recovery model.

3.3. The catalyst module

We have previously identified the catalyst or reconfig-

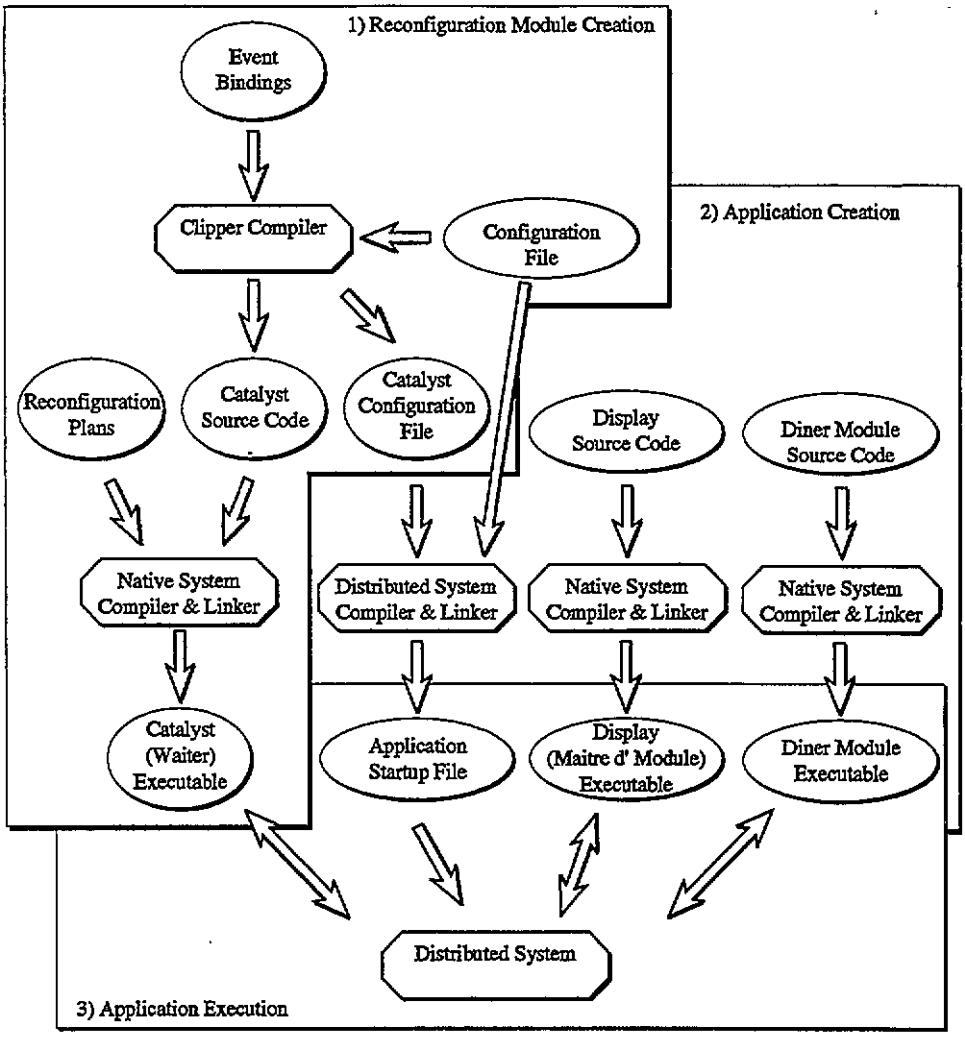

Figure 2. A pictorial representation of the creation and execution of the diners example as a Clipper application.

uration module. This is the central change agent that is dedicated to processing the application configuration changes embodied within the reconfiguration plan.

3.3.1. Application model. The catalyst maintains a model of the application in its currently executing state. This provides access to all structural components (bindings and modules) of the application in the reconfiguration plan. Since we perform no analysis of reconfiguration plans, we are not aware of the type of configuration changes being made, and hence, what components must be accessible to each plan. This would be a concern if plans were allowed to execute concurrently, since concurrent execution of plans operating on the same application components would result in a resource conflict. At this time Clipper is restricted to serial execution of plans.

In the diners example, the plan attached to the "arrival" event controls additions of all new diners. This requires that it have access to any pair of directly connected diners, though it does not need access to the display module.

The model maintained by the catalyst is separate from the distributed environment's representation of the application. There are several advantages to this approach.

- A separate model gives us more flexibility in deciding the semantics of a reconfiguration plan. By maintaining our own representation, the plan implementor is not constrained by the methods of the distributed system for gaining capability to application components. In some cases, information about the application may not be present within the distributed system. This would limit the reconfiguration possible using the language.

- It minimizes the effect of the plan executions on reconfiguration performance. Queries to the distributed system involve resources that may otherwise be devoted to the application. By keeping separate data, we reduce the load on the system.

- Better application representation. Using the Clipper application model we can infer more about the application than possible through the distributed system. Some examples are the number of modules running on a given machine, or the bindings between two modules.

Table 4. The Configuration specification for the dining philosophers example.

Module definitions	Application definition

service "specialdiner" : {
  implementation : { binary:"diner.out" }
  algebra : { "STATUS=special" }
  function "left_fork" : {} returns {}
  client "right_fork" : {} accepts {}
  client "left_req" : {} accepts {}
  function "right_req" : {} returns {}
}
service "diner" : {
  implementation : { binary : "diner.out" }
  algebra : { "STATUS=static" }
  function "left_fork" : {} returns {}
  client "right_fork" : {} accepts {}
  client "left_req" : {} accepts {}
  function "right_req" : {} returns {}
}
service "display" : {
  implementation : { binary : "display.out" }
}
	orchestrate "table" : {
	tool "frontend" : "display"
	tool "Jim" : "specialdiner"
	tool "Jack" : "diner"
	tool "Christine" : "diner"
	tool "Liz" : "diner"
	bind "Jim left_fork" "Christine right_fork"
	bind "Jim left_req" "Christine right_req"
	bind "Christine left_fork" "Liz right_fork"
	bind "Christine left_req" "Liz right_req"
	bind "Liz left_req" "Jack right_req"
	bind "Liz left_fork" "Jack right_fork"
	bind "Jack left_fork" "Jim right_fork"
	bind "Jack left_req" "Jim right_req"
	}

3.4. Catalyst module creation

Figure 2 shows the use of Clipper in constructing the new dynamically reconfigurable diners application. We have separated the diagram into three parts: creation of the reconfiguration module using the Clipper and native system compilers, creation of the remaining application components using the distributed system and native system compilers, and execution of the application. There are two application configuration descriptions used to create the catalyst.

- The initial configuration of the application, the *static* description. We use this to initialize the application model at the begining of execution. For the distributed system on which our language is currently implemented, this description is contained in the configuration specification shown in table 4.

- The reconfiguration modes of the application, or *dynamic* description. These are the plans which describe configuration changes to the application. Table 1 shows the reconfiguration plan for our example.

3.4.1. Parsing the static description. The configuration specification file consists of two parts. The first contains the *module descriptions*, or declarations of the module types that are available for instantiation in the application. The second contains the *application description*, or bindings and modules that make up the application at the start of execution. We extract from the module description section of the configuration specification the module definitions for creating model object types. From the application description we get the bindings and modules that are described in the model at startup.

3.4.2. Assembling the catalyst module. The catalyst executable consists of the model definition, model initialization function (needed to synchronize the model with the application at startup), and the main function. The main function contains the model initializer call and the event handling loop. The initialization function is made up of statements for initializing the event handler and statements for initializing the application model. The event and binding declarations extracted from the bindings file initialize the event handler. The model object and bind statements derived from the configuration specification initialize the model. To simplify construction, we have limited the amount of code that is generated by the Clipper compiler to the initialization function and model object definitions. The remaining definitions are contained in the Clipper library, which is linked with the user's plans and the initialization function definition at compile time.

3.4.3. Execution of the catalyst module. The catalyst module is started when the application begins execution. Event handlers execute when the event generator they are bound to sends a signal to the catalyst module. Execution order of the event handlers bound to the same event generator follows the order in which they are declared in the bindings file. Processing event generator signals is serialized at the catalyst. New events queue in the order they are received, until all event handlers have executed for the current event.

3.5. Execution of the reconfiguration plan

The module display is the event generator signalling the user's desire for a new diner to be added to the application. After the event generator receives the names of a new diner and two adjacent diners, the module sends an "arrival" event to the catalyst to initiate the configuration change. The catalyst then processes any

plans that are bound to the "arrival" event (only one for this example). Once the plan bound to this event is started, we gain access to the two adjacent diners and a new diner with the diner constructors for objects `oldleft`, `oldright` and `newdiner`. Following this, we change newdiner's STATUS attribute to indicate that this module has been started dynamically. If this attribute is not set, the new module will not synchronize with the two adjacent diners. Bindings between modules `oldleft` and `oldright` are removed with **unbilink** commands. The fork bindings are attached between the two existing modules and the new module using **bilink** commands. Since there are no other attributes that need to be initialized, the module can now be added to the application. Model objects `oldleft` and `oldright` are signalled to dump their state information to `newdiner` through interfaces "reconfig_right" and "reconfig_left" respectively.

4. Discussion

In this section we discuss some of the issues that arose as a result of our implementation of Clipper.

Atomicity. So far we have not shown the sequence of low-level reconfiguration commands executed by the reconfiguration operations in Clipper. It has been assumed that the sequence of events executed by these commands is prevented from causing failure by a set of preconditions. However, in many cases there is the possibility that the configuration of the application may be altered during the execution of a reconfiguration command, creating unpredictable results. If the reconfiguration commands are not executed atomically, the final configuration is undetermined. For example, if access to attribute values involved spawned processes, there is the possibility of `newdiner` in the diners example being added to the application before its STATUS attribute is modified. This would cause the module to starve. Much of our effort in the prototype goes into addressing the issue of atomicity of operations.

Transformations that are not specified by the reconfiguration programmer in Clipper cannot be guaranteed successful. At this time, our prototype assumes there is only one agent responsible for performing reconfiguration operations. In the diners application, one transfer function was specified for the application, that of adding a new diner to an application containing two diner modules directly bound. This reconfiguration is not protected against the condition of a specified diner not existing. Dynamic checking of reconfiguration commands will prevent application failure locally, but will not discourage the propagation of an error through the plan. Let us assume the user specifies a non-existent left module. A new module will be created at the declaration of `oldleft`. Since this module is not part of the application, **unbilink** will not attempt to unbind it from `oldright`, **bilink** will not attach it to `newdiner`, and **dumpstate** will not signal it to synchronize with `newdiner`. The result will be an application in which `newdiner` is partially bound and starving.

Event handling. In the current implementation of Clipper, the plan module is not represented in its application model. The intent of this is to satisfy the ideal that the catalyst is an *external* change agent, independent of the application configuration, and therefore not part of the application model. This means that we cannot explicitly bind modules to the catalyst, nor can we start a new catalyst using the existing one. With only application based event generators available, we had to be sure the event generator that initiated the addition of a new diner was not a component that would be reconfigured. If it was, then the plan module would not be able to receive additional events after the first new diner was added. The main difficulties in implementing the dining philosophers problem were transmitting reconfiguration events to the catalyst, attaining capability to dynamic objects, and synchronizing new components with the existing application.

Event passing network. During our prototyping we introduced an event communication network for organization of reconfiguration events for improved synchronization of event generators. This network uses the framework of Polylith Multicast [2]. The design considerations of the network are similar those of the Distributed Object/Concurrent-Thread system (DO/CT), an event handling system for asynchronous and synchronous events by Menon [10] *et al.*

Implementation of the event network using the language semantics of multicast is as follows: Events are represented by message type. When an event generator activates, it multicasts a message to the catalyst of a type appropriate for that event. The catalyst remains blocked until it receives a multicast message. When a message is received, the catalyst executes the plans that have been declared as handlers for that type of message.

Multicast message passing provides a seamless interconnect between many event generators and the catalyst. It also simplifies the implementation of multiple reconfiguration agents running concurrently in the same application. The latter benefit is not one we exploit at the moment because of considerations involving synchronization of configuration modes in the application with multiple, simultaneous reconfiguration processes. This involves reconfiguration control beyond our current design objectives.

Model structure query language. More flexibility in gaining access to model components could be realized through the introduction of model access methods based on the structural or component nature of the application. For example, we may be interested (for load balancing purposes) in moving all of the modules on machine 'A' to machine 'B'. To do this, we search the model and replace all modules with a machine attribute of 'A' with modules with machine attribute of 'B'.

Related work. This work directly builds upon prior research performed at the University of Maryland. Previously, Surgeon (built upon Polylith) provided a

general mechanism for performing reconfiguration [6]; its focus was on making run-time capabilities available in widely heterogeneous environments, where the reconfiguration activities were invisible to application programmers. This largely paralleled (and to some extent also built upon) work in the Conic project [8] as previously reported. One of the differences was degree of transparency to the programmer. The Surgeon system placed great emphasis upon making application reconfigurations invisible to the programmer, whereas Conic placed requirements upon the designer in order for the application to be reconfigurable. But this was also a tradeoff, since the Conic work had greater emphasis on characterizing correctness and consistency conditions in the reconfiguration operations. This included a great deal of analysis to identify a 'stable state' within which reconfiguration steps would be acceptable, and hence produced a system with more safety conditions. The Surgeon system opened a much broader set of commands (and states within which those commands could be performed) to the reconfiguration programmer, but without necessarily maintaining as strict a set of safety conditions.

In Surgeon, the reconfiguration programmer (as opposed to the application programmer) still had several manual steps to perform in order to prepare to reconfigure a running application. As noted earlier in this paper, this was the motivation for our current work. The Clipper language reflects our desire to provide a simple and compact notation for programmers to characterize large classes of reconfiguration steps. Clipper might be though of as a high level environment for interacting with Surgeon, although this image would not capture the design effort of automatically generated stubs and catalyst modules. In the same way that Surgeon has evolved into the current work, the Conic effort eventually motivated Rex and then Regis [9], a powerful environment. In some respects, the derivation of Regis contains many of the desirable characteristics of the earlier Surgeon, and similarly our own work now reflects a step towards increased levels of formalism as originally inspired by Conic.

Other systems have emerged to assist programmers in performing reconfiguration operations dynamically. The Schooner system [7] provides an efficient run-time mechanism for reconfiguration of the application within the scientific computing domain, while the Durra [1] system covers real-time computing reconfiguration concerns. Each of these systems provides efficient methods for reconfiguration but with fairly strict assumptions limiting their application domain. More recent developments include environments which provide an object oriented approach to the implementation of reconfigurable applications [13], as well as supporting reconfiguration mechanisms such as those found in Surgeon or Rex.

Further improvements. We would like to create a more comprehensive set of plans that are triggered by changes to application components based on the type of these components. These 'plans' already exist in C++ in the form of object constructors and destructors. An example of their use in the diners application is in the addition or removal of a diner. A new diner is always bound to a left and right diner. Consequently, its constructor is a function that has two model object parameters. When the constructor is called, it initializes its object's STATUS attribute and binds its object between the diner object arguments passed to it. Similarly, a diner object's destructor is a function that disconnects its object and reconnects its object's neighbours to each other. We discourage reconfiguration programmers from creating constructor or destructor plans in the current implementation of Clipper in order to protect the underlying structure of the application model.

5. Conclusion

Using the specifications presented in section 2 we have created a simple reconfiguration language for specifying application configuration transitions in terms of event generators signalling change and event handlers describing change. The plan syntax provides an intuitive way of manipulating the application structure through a set of heterogeneous objects making up a model of the application. We promote correct execution of reconfiguration using two methods: type checking for static analysis (using the C++ type checking facilities), and dynamic checking for correctness of reconfiguration operations (by imposing constraints on changes of the application model). We specify application configuration transitions as a separate component of the application specification; these are incorporated into an external change agent executed with the application. This process allows the reconfigurations to be handled in a manner that minimizes the changes to the application, and insulates the plan writer from the distributed platform.

References

[1] Barbacci M, Doubleday D, Weinstock C, Gardner M and Lichota R 1992 Building fault tolerant distributed applications with Durra *Proc. Int. Workshop on Configurable Distributed Systems (London)* (London: IEE) pp 128–39

[2] Chen C, White E and Purtilo J 1993 A packager for multicast software in distributed systems *Proc. 5th Int. Conf. on Software Engineering and Knowledge Engineering (San Francisco)* (Skokie, IL: Knowledge Systems Institute) pp 612–21

[3] Ellis M and Stroustrup B 1991 *The Annotated C++ Reference Manual* (New York: Addison-Wesley)

[4] Endler M and Wei J 1992 Programming generic dynamic reconfigurations for distributed applications *Proc. Int. Workshop on Configurable Distributed Systems (London)* (London: IEE) pp 68–79

[5] Etzkorn G 1992 Change programming in distributed systems *Proc. Int. Workshop on Configurable Distributed Systems (London)* (London: IEE) pp 140–51

[6] Hofmeister C, White E and Purtilo J 1993 Surgeon: a packager for dynamically reconfigurable distributed applications *Software Engng J.* **8** (2) 95–101

[7] Homer P and Schlicting R 1994 Configuring scientific applications in a heterogeneous distributed system *Proc.*

2nd Int. Workshop on Configurable Distributed Systems (Pittsburgh, PA) (Los Alamitos, CA: IEEE) pp 159–71

[8] Kramer J and Magee J 1990 The evolving philosophers problem: dynamic change management *IEEE Trans. Software Engng* **16** 1293–306

[9] Magee J, Dulay N, and Kramer J 1994 A constructive development environment for parallel and distributed programs *Proc. 2nd Int. Workshop on Configurable Distributed Systems (Pittsburgh, PA)* (Los Alamitos, CA: IEEE) pp 4–11

[10] Menon S, Dasgupta P and LeBlanc R 1993 Asynchronous event handling in distributed object-based systems *Proc. 13th Int. Conf. on Distributed Computing Systems (Pittsburgh, PA)* (Los Alamitos, CA: IEEE) pp 383–90

[11] Purtilo J 1994 The Polylith software bus *ACM TOPLAS* **1**

[12] Purtilo J and Hofmeister C 1991 Dynamic reconfiguration of distributed programs *Proc. 11th Int. Conf. on Distributed Computing Systems (Arlington)* (Los Alamitos, CA: IEEE) pp 560–71

[13] Schmidt D and Suda T 1994 The service configurator framework: an extensible architecture for dynamically configuring concurrent multi-service network daemons *Proc. 2nd Int. Workshop on Configurable Distributed Systems (Pittsburgh, PA)* (Los Alamitos, CA: IEEE) pp 190–205

[14] Young A and Magee J 1992 A flexible approach to evolution of reconfigurable systems *Proc. Int. Workshop on Configurable Distributed Systems (London)* (London: IEE) pp 152–63