Support for Consistency-Preserving Dynamic Reconfigurations in Distributed Systems

Markus Endler
GMD Research Laboratory at the University Karlsruhe
Vincenz-Prießnitz-Str 1, D-7500 Karlsruhe, Germany
Email: endler@karlsruhe.gmd.de

Abstract

In this paper we present a model with its linguistic and operational support for implementing consistency-preserving reconfigurations of distributed applications. Our model extends the Configuration-programming approach[2], where a distributed program is implemented at a *programming* and a *configuration level* and where dynamic changes are expressed only at the configuration level, i.e. as modifications of the connectivity structure among the parallel executing components.

In our approach, besides the communication interface, every component of the application owns a *reconfiguration interface*, which defines the reconfiguration-specific interactions between the application processes and the reconfiguration processing. With such an interface, application and reconfiguration concerns can be well separated and existing applications can be easily upgraded to support consistency-preserving reconfigurations.

1 Introduction

Support for dynamic reconfiguration is becoming more and more important for distributed systems, especially in long-living and real-time applications where off-line maintenance is either impossible or too costly.

The most successful approaches for describing and implementing dynamic reconfiguration have been the Configuration-programming approaches[1, 2, 4, 5], where a distributed program is implemented at two levels: at the *programming level* the application's functionality is encoded into components with a communication interface, and at the *configuration level* instances of those components are created and interconnected for interaction. In these approaches, dynamic reconfiguration is performed only at the configuration level, i.e. as a change of the components' interconnection structure.

Very few approaches, however, have addressed the problems of preserving the consistency of the application during a dynamic change. In [3] it is assumed that the application program is already implemented in a reconfiguration robust way, i.e. that it can go into a stable and reconfiguration-suitable state whenever a reconfiguration is to be performed. Since this is not the case for most of the existing applications, a method for adding such capability in a modular way is required.

In this paper, we present a configuration-based model with its linguistic and operational support for implementing consistency-preserving reconfigurations for existing distributed applications.

Within our approach, every component of the application has a *reconfiguration interface*. This interface specifies *events*, *state predicates* and *consistency operations*. Events associate the execution of certain commands within the component to externally visible signals. They are made visible to the configuration level through state predicates, which are used to synchronize the reconfiguration with the execution of the components to be manipulated. Consistency operations are special procedures which can be invoked from reconfiguration scripts and which define manipulations on the component's internal state, similar to the actuators in[6]. These operations have access to the whole component state by referring to its variables and labeled commands within the component's source code. Because reconfiguration interfaces can be designed after the application has been implemented, our approach liberates the application programmer from having to consider dynamic reconfiguration during the design of the component algorithms. However, for upgrading an application, all the components have to be recompiled together with their reconfiguration interface.

At the configuration level, one can design dynamic changes that depend on the occurrence of reconfiguration events and that invoke consistency operations at some components. This gives a means for synchronizing the dynamic reconfiguration with the execution of the involved application components and for keeping the component's internal state consistent with the changes produced by the reconfiguration.

0-8186-2755-7/92 $03.00 © 1992 IEEE

In the following we explain the linguistic and operational support for our approach, using the example of a telephone exchange implementation, adapted from [4]. Section 2 introduces the structure of the telephone exchange. Section 3 presents the language used to implement the consistency-preserving reconfigurations of the example and section 4 explains the elements of the reconfiguration interface. In section 5 we explain the required operational support and in section 6 we derive some conclusions.

2 Telephone Exchange

The following example shows the implementation of a telephone exchange as a composite component in the configuration-programming approach. Other than primitive components (or program components), which encode the application-specific algorithms and are implemented in some imperative programming language, composite components are built from interconnected instances of other components, and contain a set of programmed changes to dynamically manipulate their internal connectivity structure. Both program and composite components own a communication interface consisting of ports or portsets, through which typed asynchronous or synchronous messages are transferred. Portsets are groups of functionally related ports to which the connection and disconnection operations apply as a whole.

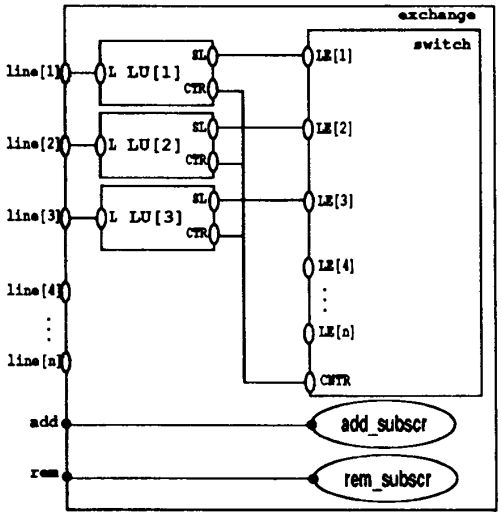

Figure 1: Structure of the Telephone Exchange

For every telephone device connected to a portset *line[i]* of this exchange, there is a line unit (*LU*) (instance of program component LineUnit) responsible for processing the speech and control signals from and to the device, as well as the call charges for the corresponding phone number. The other instance is *switch*, a composite component depicted in Fig. 2, which builds and removes speech connections between the line units upon requests through its portset *CNTR*. Component *Exchange* further contains two consistency-preserving changes *add_subscr* and *rem_subscr*, which create and remove line units as new telephone devices are connected or disconnected from *exchange*. Such changes can be invoked by any component connected to them, i.e. in our example by any instance connected to ports *add* and *rem* of *exchange*.

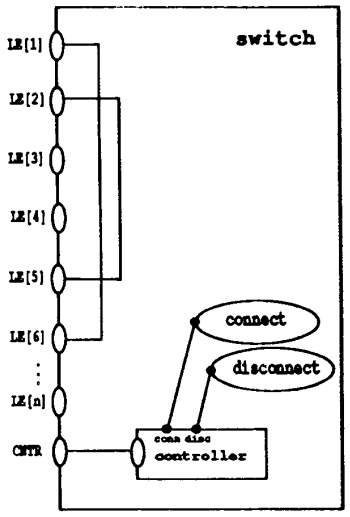

Figure 2: Structure of the Switch

Figure 2 shows composite component *switch* consisting of the instance *controller* (program component) and the programmed changes *connect* and *disconnect*. These changes link or unlink two portsets *LE* (line entry) of *switch*, whenever this is requested by any of the line units connected to one of these portsets. Requests for a telephone connection are forwarded to instance *controller*, which then invokes either *connect* or *disconnect* with the index of the portsets to be linked. Note that for *controller* to be able to correctly manage the change calls, its internal data structures must reflect the current connectivity within *switch*. Therefore, in this case a change at the configuration level will always require a change in the component's internal data.

3 Reconfiguration Language

At the configuration level the linguistic support of our approach is given by an extension of the reconfiguration language Gerel[7]. Gerel (*Generic Reconfiguration*

*L*anguage) is a high-level configuration and reconfiguration language, used to implement both the initial configuration and the programmed changes within composite components. The extensions comprise the inclusion of commands for synchronizing with component executions (**when**) and for executing consistency operations (**exec**). In what follows, we give a short presentation of the extended language.

Gerel consists of a set of basic reconfiguration commands, a set of structured commands and a declarative, logic-based query language called *Gerel-SL*, which is used to describe expected structural properties of configuration objects (e.g. component instances and their interaction points).

3.1 Basic Commands

The following are Gerel's basic reconfiguration commands, which define the creation and removal of component instances and the communication connections between them.

create *instance* **of** *type* [(*arguments*)] [**at** *location*]
delete *instance*
link *port port*
unlink *port port*
bind *portset portset*
unbind *portset portset*

3.2 Structured Commands

Gerel's structured commands define the control flow of reconfiguration actions, their synchronization with execution of sub-instances, object selection and the call of consistency operations:

select *gs* : *selection-formula* **do** *commands* **end**
forall *gs* : *selection-formula* **do** *commands* **end**
iterate *i* **in** [*low*:*high*] **do** *commands* **end**
when *condition* **do** *commands* **end**
exec *consist.-operation*

where **select** and **forall** are the commands implementing the selection mechanism of Gerel: according to *selection-formula* the set of objects to be manipulated in *commands* is determined. If this set consists of more than one object, **select** randomly chooses one object and **forall** executes *commands* for each object in the set.
Iterate is a simple for-loop, useful for manipulating elements within arrays of components, ports or portsets.
When delays execution of *commands* until *condition* (a state predicate) is satisfied and **exec** calls a consistency operation at a given component.

3.3 Generic Symbols

Gerel supports the definition of reconfiguration variables, called generic symbols. Generic symbols can only have names of configuration objects as values. Corresponding to the four kinds of configuration objects (component instances, component types, portsets and ports) Gerel supports four types of generic symbols, *inst*, *type*, *port* and *pset*, respectively. Additionally, the range of a generic symbol of type inst, pset or port can be further constrained to a certain concrete object type, as shown in the following example:

```
symbol
  i:  inst;     /* instances of all types */
  n:  inst abc; /* instances of type abc */
  s:  pset xyz; /* portsets of type xyz */
  m:  type      /* all component types */
```

3.4 Gerel-SL

Gerel-SL is a first order logic language within *Gerel* used to describe structural properties of component instances, component types, ports and portsets within a configuration. Gerel-SL can be used both for specifying selection formulas and for expressing reconfiguration conditions.

Gerel-SL formulas are built using the logical negation (keyword **NOT**), the logical conjunction and disjunction, denoted by symbols **&** and |, and the universal and existential quantifiers, respectively denoted by the keywords **FA** and **EX**.

Besides this, Gerel-SL supports the use of a rich set of primitive predicates and functions, which express properties and relations between configuration objects. Because most of the primitives are required for different kinds of objects, we use prefixes c_, ct_, s_ and p_ to denote, respectively, the variants for component instance, component type, portset and port. In the following the parenthesis expression lists the available variants of each primitive:

Predicates	
(c,s,p)_linked(o_1,o_2)	objects o_1 and o_2 are linked
(c,s,p)_exists(o)	object o exists
(s,p)_free(o)	object o is not connected
Functions:	
invoker()	instance calling the change script
(c,s,p)_type(o)	the type of o
(s,p)_owner(o)	the instance to whom o belongs
(s,p)_def(o)	the definition name of object o
(c,s,p)_index(o)	the index of an array instance

3.5 Change Structure

A change implements an atomic reconfiguration transaction and is defined in a ***change script*** with the following structure:

```
change name (parameters)
  symbol
      generic symbol declarations
  condition
      reconfiguration precondition
  execute
      reconfiguration actions
end.
```

where the **symbol** and **condition** sections are optional and *reconfiguration actions* is only executed if *reconfiguration precondition* is satisfied.

For specifying the initial configuration, a special change called **initial** is used, which is automatically invoked whenever a new instance of the composite component is created.

In the following we show the Gerel implementation of the two changes of *exchange*. They are consistency-preserving in the sense that no ongoing telephone call is disturbed or interrupted. While *add_subscr* trivially preserves consistency, change *rem_subscr* has to be synchronized with the execution of the line unit to be removed.

```
change add_subscr (entry, phone_nr:  integer);
symbol X: inst LineUnit;
condition p_free( own.line[entry]);
execute
 exec switch.register(entry, phone_nr);
 create X of LineUnit (phone_nr);
 bind X.L own.line[entry];
 bind X.SL switch.LE[entry];
 bind X.CTR switch.CNTR
end.
```

Change *add_subscr* creates a new line unit whenever a new telephone device that is connected to the exchange. Before creating and interconnecting the new line unit, however, *add_subscr* executes the consistency operation *register* of instance switch, for registering the new device and assigning it to a free line of the exchange.

```
change rem_subscr (entry, phone_nr:  integer);
symbol k:  inst LineUnit;
     p,q:  pset;
execute
 select k:  s_linked(k.L, own.line[entry]) do
  when NOT k.talking do
   exec k.close_account;
   exec switch.unregister(entry);
   forall p:  s_owner(p)=k do
    forall q:  s_linked(p,q) do unbind p q end
   end
   delete k
  end
 end
end.
```

Change *rem_subscr* removes line units whenever telephone devices are to be disconnected from the exchange. This, however, is only done when the line unit is not involved in an ongoing conversation. If this is the case, *rem_subscr* waits until conversation is finished and only then performs the reconfigurations. According to the above script, *rem_subscr* first selects the line unit to be removed (*k*) and then waits until it is not in state *talking*. Only when this state is reached, it requests the final bill charging operation *close_account* and unregisters the phone number in instance switch. At last, the line unit and all its connections are removed.

4 Reconfiguration Interface

The *Reconfiguration interface* defines the way adjacent levels of the system configuration interact and synchronize during a reconfiguration. Other than the communication interface mentioned in section 2, which is only used for transmitting application-specific data among components, the reconfiguration interface specifies the synchronization and interaction between the reconfiguration- and the application-specific processing.

A reconfiguration interface consists of *events*, *state predicates* and *consistency operations*. It has to be provided for every component within the application and is implemented in an extended version of the language used to implement the component. The required extensions can be defined for every programming and configuration language and can be implemented by language pre-processors.

4.1 Events and Consistency Points

Events represent abstractions of internal component states. They signal the execution of certain commands within the component to its environment, i.e. to the configuration level directly above. Based on the information about the occurrence of events, changes can be delayed until a certain component state has been reached.

In the following we show the events of component LineUnit and how they are defined in its reconfiguration interface.

```
event used1, used2, free;
```

Such events denote both a program point within the component's source and a boolean variable, which is set (true) whenever this program point is executed.

Whenever events are defined for program components, they must be related to statements of the components' source code. This is done by attaching the event name (surrounded by symbol #) to the corresponding source code statement, as shown in the following skeleton of the C-based implementation of LineUnit. The event is then *announced* immediately *after* the marked statement is executed, causing the previous event to be reset.

```
component LineUnit( my_nr) = {
 int my_nr;
 interface declaration
 for (;;) {
  select
   #used1#accept inL(offhook) => {
    call outL(line_tone); accept inL(dial_nr);
    call outCTR(conn,my_nr,dial_nr);
    accept inCTR(lineok);
    if lineok try to reach other party }
   #used2#accept inCTR(ring_req) => {
    call outL(ring_req);
    wait for offhook from own device }
   end

   /* speech phase */
   while (lineok==1){
    process speech signals and exit speech phase
    if any of the parties signals an onhook }

   #free#
    process call charge;
   #...
  } /* endfor */
 } /* LineUnit */
```

Consistency points define the states of a component in which its internal data has reached a consistent state and where the component's processing can be interrupted for the purpose of a reconfiguration. All events of a component are also consistency points. In program components one can define further consistency points (that are not events). This is done by attaching symbol # to the statement after which the component can be interrupted. In LineUnit an extra consistency point was set at the end of the (main) loop body.

Within composite components, events can be attached only to programmed changes, for signaling when the component's internal structure, i.e. its internal state, has being modified by a change. Since all changes within a composite component are atomic and a consistent state is reached whenever no change is in execution, in this case there is no need for defining further consistency points.

With a language preprocessor, consistency points and events are mapped onto global variables, labels and standard procedures for interrupt processing.

4.2 State Predicates

State predicates (or predicates) are the only means of presenting component states to higher levels of the configuration. They must also be declared in the reconfiguration interface and consist of boolean expressions over event announcements and over language-specific relational or logical expressions in the component's implementation language. With the latter, one can relate state predicates to the contents of component internal variables.

The following shows the definition of predicate *talking* of LineUnit, used within change rem_subscr.

```
pred talking == used1 || used2
```

It says that *talking* is true whenever event *used1* or event *used2* is announced.

Other than the announcement of events, which goes along with the component's execution, the evaluation of predicates is activated by a change. Activated predicates are evaluated every time the corresponding component announces an event, causing a momentary suspension of its execution. Only if any of its activated predicates becomes true, then the component definitely looses control over its execution, i.e. it is forced to stop and control is given to the change that activated the predicate. This control is given back as soon as the corresponding reconfiguration command is finished. The only exceptions are Gerel's commands *select* and *forall*, in which control is retained only for the selected components.

Whenever a predicate is activated it is evaluated on the basis of the last announced event. The component, however, continues execution until the next consistency point and then stops or resumes computation depending on the current predicate value.

For every predicate definition, a system-known port (for its activation) and an evaluation procedure are generated in the component, when recompiling it with the interface.

4.3 Consistency Operations

Consistency operations are the means for manipulating component internal data and control flow from outside. They are imperative for consistency preserving reconfigurations, since often the component's connectivity and its internal state are closely related, such that changes in the connectivity must be followed by internal updates.

Because the internal state of a composite component is its current configuration, i.e. the connectivity structure within it, consistency operations for this kind of component must be reconfigurations as well. Like predicates, consistency operators are also programmed in the component's source language, i.e. in our case, in extended Gerel.

Therefore consistency operations may themselves refer to state predicates and call consistency operations of the composite component's sub-instances. This gives a means for defining synchronizations and state manipulations across many levels of the configuration hierarchy.

In the following we show consistency operator *unregister*, defined as a Gerel change within the reconfiguration interface of switch.

```
oper unregister ( entry:  integer);
symbol p:  pset
execute
 when controller.no_request do
  forall p:  s_linked(own.LE[entry], p) do
   unbind p own.LE[entry]
  end
  exec controller.rem_user(entry)
 end
end.
```

Change *unregister* first waits until instance *control* is in a state where no new connections are being requested (predicate *no_request*). Then it removes all connections to the portset (*own.LE[entry]*) of the removed line unit and calls the consistency operation *rem_user*, by which it modifies the data within *control*, to reflect the changes in the connectivity.

Consistency operations for program components take the form of procedures which have access to all global definitions within the components source. Assuming that *control* is a program component and that the information about the currently used entries of *switch* and their current connections are stored, respectively, in arrays *in_use* and *conn_with* (both of dimension max), following would be the C-based implementation of *controller*'s consistency operator *rem_user*.

```
oper rem_user(entry)
int entry;
{
 in_use[entry]=0;
 for (i=1; i<max; i++)
  if (conn_with[i]==entry) conn_with[i]=0;
 }.
```

5 Operational Support

The operational support required by our model is mainly the one provided by the runtime environment of reconfigurable configuration-based systems. In those systems, programmed changes are usually compiled into invisible system processes (*change processes*) which manage the execution of changes. These change processes have the knowledge of the location of all the managed instances and their ports. In our model, for every composite component a single change process is generated for coordinating the execution of all programmed changes within the component. For distributed composite components, which have its constituting instances running on different machines, the change process knows the network-wide location of every sub-instance.

In order to support an efficient evaluation and execution of state predicates and consistency operations in distributed composite components, predicate evaluation, interrupt management and the execution of consistency operations should be implemented locally, i.e. at every component, leaving only their activation to be remote.

This can be done by allocating an extra (system defined) port at the component for every state predicate and consistency operation defined. These ports are used to announce the activation of a predicate and pass the arguments for a consistency operation. Like communication ports, these ports are known to the change process responsible for the enclosing composite component. Whenever a change containing a state predicate of a remote instance is invoked, this change process sends a message to the component informing it about the predicate activation. After this, the change process blocks waiting for a response from the component. The component, on the other hand, checks for a predicate activation whenever it announces an event or reaches a consistency point. When any of the activated predicates becomes true, the component acknowledges the predicate activation and suspends normal execution waiting either for the execution of a consistency operation or a predicate deactivation. Only then it resumes normal computation. A similar protocol is used when consistency operations are activated and their execution is delayed until a consistency point is reached.

The only additional support required by our model is therefore in the language preprocessing. Within program components, at every declaration of an event or consistency point, a label and a jump to a system generated code for the interrupt management have to be produced. The code of the predicate and consistency operation evaluation (in the reconfiguration interface) is then pasted into this interrupt management section, resulting in a component program which is sensitive to activation of its predicates and operations.

For composite components, similar transformations are done when generating the change process and consistency operations are transformed into additional programmed changes.

6 Summary and Discussion

In this paper we presented a model for the implementation of consistency-preserving reconfigurations. It is centered around the concept of a reconfiguration interface, defining the synchronization and the interaction between the reconfiguration- and the application-specific computations. This model offers a series of advantages, which include:

It supports the strict separation of configuration and programming concerns, which is the main principle underlying the configuration-programming approach. Therefore, besides inheriting all the advantages of this approach, like e.g. compositionality, scalability and sup-

port for language heterogeneity, we are able to treat consistency-preservation in different levels of abstractions (according to the level of configuration being considered) and in an application independent way (although its actual realization is application dependent).

With the concept of a reconfiguration interface change-specific synchronizations and interactions are made explicit and the system behavior can be better understood. Moreover, changes can be better shaped to application-specific consistency requirements, which is more difficult when standardized change management policies, as defined in [3], are used.

Our approach is language independent in the sense that the required support can be implemented for any imperative programming language and any reconfiguration language having commands similar to the *when* and *exec* of Gerel.

Our model supports a simple and modular extension of existing configuration-based applications, for implementing consistency-preserving reconfigurations. This extension is modular because the definition of reconfiguration interfaces imply in very few changes to the component's original implementation (actually, only labeling).

However, we also recognize some of the weaknesses of our approach. Our model does not give a solution to all sorts of consistency problems. In particular, interaction problems caused by the blocking of components are assumed to be properly prevented at the programming level, e.g. by instrumenting message passing actions with time-outs.

Besides this, in our approach changes are more complex since they include explicit synchronizations and consistency operation calls. The approach also presumes that the change programmer has some knowledge of the functionality of the components to be manipulated by the change. On the other hand, the definition of the reconfiguration interface requires a precise specification of the kinds of reconfiguration which will be applied on each component, what may be a problem when changes are added or redefined. Our model, therefore, induces a re-engineering methodology which requires intense interaction between the programmer of the change and that of the component to be manipulated.

Future Work

We are now investigating whether it is necessary to provide histories of events for defining predicates which express ***knowledge*** of the component's previous execution.

Another topic of investigation is related to the definition of state predicates for composite components. The question is whether it is necessary and effective to define such predicates in terms of the predicates of constituting instances. This would not only make predicate evaluation more inefficient, but also directly affect the execution of the application. Furthermore, it would be more likely to cause deadlock situations in the sub-instance's interaction.

Acknowledgements

Special thanks go to my colleagues at GMD Karlsruhe for their valuable comments and suggestions. Acknowledgement is also made to CEC in the REX Project for the financial support.

References

[1] M.R. Barbacci and J.M. Wing. Durra: A task-level decription language. Tech. Report CMU/SEI-86-TR-3, Software Engineering Institute, Carnegie Mellon University, 1986.

[2] J. Kramer. Configuration programming - a framework for the development of distributable systems. In *Proc. IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro90)*, Tel Aviv, Israel, May 1990.

[3] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, SE-16(11):1293–1306, November 1990.

[4] J. Kramer, J. Magee, M. Sloman, N. Dulay, S.C.Cheung, S. Crane, and K. Twidle. An Introduction to Distributed Programming in REX. In *Proceedings of the ESPRIT Conference (ECE 91)*, 1991.

[5] R.J. LeBlanc and A.B. Maccabe. The design of a programming language based on connectivity networks. In *Proc. of the 3rd Int. Conference on Distributed Computing Systems*, pages 532–541. IEEE, 1982.

[6] K. Marzullo, R. Cooper, M.D. Wood, and K.P. Birman. Tools for Distributed Application Management. *IEEE Computer*, 24(8):42–51, August 1991.

[7] J. Wei and M. Endler. A configuration model for dynamically reconfigurable distributed systems. In B.D. Shriver, editor, *Proceedings of the 24th Annual Hawaii International Conference on System Sciences*, pages 265–274. IEEE Computer Society Press, January 1991.