Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @DIRECT®

Science of Computer Programming 48 (2003) 43–65

Science of Computer Programming

www.elsevier.com/locate/scico

Dynamic configuration management in a graph-oriented Distributed Programming Environment

Jiannong Cao[a,*], Alvin Chan[a], Yudong Sun[a], Kang Zhang[b]

[a]*Software Development and Management Lab, Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong*

[b]*Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083-0688, USA*

Received 11 April 2001; received in revised form 11 July 2002; accepted 27 August 2002

Abstract

Dynamic configuration is a desirable property of a distributed system where dynamic modification and extension to the system and the applications are required. It allows the system configuration to be specified and changed while the system is executing. This paper describes a software platform that facilitates a novel approach to the dynamically configurable programming of parallel and distributed applications and systems. This platform is based on a graph-oriented model and it provides support for constructing reconfigurable distributed programs. We describe the design and implementation of a dynamic configuration manager for the graph-oriented distributed programming environment. The requirements and services for dynamic reconfiguration are identified. The architectural design of a dynamic configuration manager is presented, and a parallel virtual machine-based prototypical implementation of the manager, on a local area network of workstations, is described.

1. Introduction

Programming on a distributed system for high-performance computing has been recognized as a much more difficult task than programming on a centralized system. In addition to important techniques such as task scheduling, remote invocation, interprocess communication and synchronization, configuration management of a distributed system has attracted much attention from researchers in recent years [5,11,16,17,27,28].

* Corresponding author. Tel.: +852-2766-7275; fax: +852-2774-0842.

E-mail addresses: csjcao@comp.polyu.edu.hk (J. Cao), cstschan@comp.polyu.edu.hk (A. Chan), csydsun@comp.polyu.edu.hk (Y. Sun), kzhang@utdallas.edu (K. Zhang).

0167-6423/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0167-6423(02)00168-5

The software of a distributed system is constructed from collections of functional components that are bound together to interact and cooperate with each other. The *software configuration* of a distributed system is performed by the instantiation and binding of software components as well as through the allocation of components to hardware systems [18,30]. The consolidation of software components is an inherent characteristic of a distributed system that offers a flexible environment to make modifications and extensions to the system. Computers can be added to a distributed system when they are required. Similarly, software components can be reconfigured flexibly within the system. For example, *system reconfiguration* is required for various purposes, such as the dynamic creation and replacement of instances of software components, evolution of a system with new functionality, and fault tolerance.

There are two approaches to system configuration management: *static* or *dynamic*. In the static approach, a distributed application is built statically by loading the system components onto the hosts in a distributed system. All system components should be configured at the same time. If a modification is required, the system must first be stopped, the configuration is then respecified, and finally the system is rebuilt. ADA [1] is an example of a programming language that is limited to static configuration management. This limitation is found because the component and configuration programming are not separated.

Dynamic configuration management refers to the ability to change a system's configuration while the system is running. When reconfiguration is required, a request is submitted to the configuration manager. The system configuration will be updated accordingly at run-time. Unlike the static approach, the entire system does not need to be rebuilt and only the part involved in the reconfiguration is modified. Thus, dynamic configuration management allows a new configuration to be adopted on-line, while part of the system may continue to execute. This flexibility is definitely a desirable property for a large distributed system, especially when it would be either infeasible or uneconomic to shut down the entire application or system in order to change part of it.

In this paper, we study dynamic configuration management in a graph-oriented distributed programming environment. We describe an integrated software platform, called distributed implementation of graphs (DIG) [7], which is based on a *graph-oriented* programming model. It provides a structural abstraction for reconfigurable distributed applications. As a tool for distributed programming, DIG provides a high-level logical graph construct and a collection of software facilities that support graph-oriented distributed programming. Graphs have been used in some programming languages as distributed *data structures* [10,29,32]. In contrast, in our model, graphs are used mainly as a distributed *program structure*, which consists of a collection of programming primitives that directly support computation and communication in a distributed program. Graphs can be used for naming, grouping, and configuring distributed tasks. They can also be used as a distributed computing framework for implementing uniform message passing and process coordination. Graph-oriented programming allows a programmer to concentrate on the logic of the application without having to be concerned with the low-level details associated with implementing and maintaining the underlying communication and coordination [8].

The graph-oriented model, underlying the DIG environment, facilitates a novel approach to dynamic configuration management. The management mechanism, which is presented in this paper, provides a collection of programming primitives that can be invoked from a graph-oriented distributed application to dynamically modify and extend its structure and functionality. Dynamic configuration activities can be initiated by any distributed process in the system. Dynamic configuration is achieved by executing a sequence of primitives (which is called a configuration plan). A dynamic configuration manager is responsible for validating the configuration plan and coordinating the distributed processes to execute the plan.

The rest of this paper is organized as follows. Section 2 discusses the research that is related to dynamic configuration management of distributed systems. Section 3 briefly describes the graph-oriented distributed programming model. Section 4 presents the dynamic configuration management in the graph-oriented model. Section 5 describes the dynamic configuration manager and its prototypical implementation. Section 6 concludes the paper with a discussion of future work.

2. Related work

The most important issues in dynamic reconfiguration include the properties of a program, the programming languages used and the underlying supporting environment, which facilitates arbitrary and unpredictable dynamic changes to distributed applications and systems. There has been much discussion about whether the description of the configuration should be included in the program or whether there should be a separate language to describe this aspect. Most existing programming systems integrate the description of the logical program structure into the programming languages (e.g., ADA [1], CSP [13,31], and SR [2,3]). Consequently, the interconnections among the program components are embedded at the programming level; hence, they do not support unpredictable changes in the configuration. Although some programming languages do support dynamic interconnections by allowing name passing [2] and some permit the dynamic creation of software components [1], they do not have any provision for accommodating unpredicted interconnections and unknown component types [19].

The distributed programming language and associated system that is called *ARGUS* [6] permits a large degree of dynamic reconfiguration. However, it does not enforce sufficient separation between the component programming and the configuration. The statements about the configuration are embedded in the component programs. Such an approach makes it difficult to validate changes in the configuration.

The *configuration programming* approaches [4,5,14,15,17,22,25,30] use configuration languages and associated environments to describe, construct and manage distributed software. They are based on the idea of separating the activities associated with component programming and the system configuration. They advocate having a dedicated configuration language that is separate from the language for component programming. Configuration languages specify the structure of a system as a set of components and their interconnections. Early systems [4,20] provide structuring mechanisms but they lack effective mechanisms for describing dynamic changes. They usually do not have the capability to program the configuration. The reconfiguration in these systems requires either a user's intervention or some additional mechanisms. The *CONIC* toolkit [24] provides two languages: a programming language that is used to program the functional modules and a configuration language that is used to specify the static configuration of the system. It also allows the specification of the mapping of the logical structure to the physical structure. The configuration language is a declarative language that is used only to describe the structure of a system. It is difficult to specify an unpredicted (not preplanned) reconfiguration [23]. Although CONIC supports on-line reconfiguration by allowing the submission of a new configuration, it cannot be used for some kinds of reconfiguration. For example, it cannot be used in fault tolerant systems because this kind of reconfiguration requires the intervention of an operator.

Configuration languages have been extended to describe dynamic changes in the configuration. *Darwin* is a configuration language that is used for structuring parallel and distributed programs that are deployed in the REX environment [24]. It is an extension of the CONIC environment, which is described above. In addition to representing the static structures of software and hardware, Darwin is able to represent structures that undergo dynamic changes.

Durra is a language for describing structures that is used for developing real-time distributed programs [4]. It provides a configuration language that can specify the preplanned reconfiguration of a system, which depends on the current configuration and the occurrence of certain events (usually in temporal expressions). A preplanned reconfiguration is specified by a temporal expression and a set of pairs of configurations that consist of an expected configuration and a new configuration. The system assumes that the expected configuration matches the current configuration of the system when the temporal expression is satisfied. As the conditions for a reconfiguration are specified in temporal expressions, it is not possible to specify more sophisticated conditions for a reconfiguration. Above all, the mechanism for specifying the reconfiguration becomes very cumbersome once the system allows a large number of possible configurations.

Dynamic reconfiguration raises several issues in configuration management that are related to the preservation of the consistency and integrity of applications [18,26,23]. To ensure that a change is performed in a safe manner, which results in a consistent state for the modified system, the configuration management system needs to determine a "safe" reconfiguration in which a component has to be aware of the fact that it is being reconfigured. In *Surgeon* [15], with certain restrictions, software components can be prepared to perform a reconfiguration that involves replacement and replication. ARGUS [6] can support the replacement of a module without much participation from an application.

3. The graph-oriented distributed programming model

Distributed implementation of graphs (DIG) is an integrated software platform that is based on a graph-oriented distributed programming model to provide support for constructing reconfigurable distributed programs. In a graph-oriented distributed model, a distributed program is defined as a collection of local programs (LPs) that can execute on multiple processors. The LPs represent the parallel computations. The communication between LPs is via message passing.

A graph-oriented distributed program is defined as a logical graph, $G(V,E)$, where V is a finite set of *vertices* and E is a finite set of *edges*. Each edge of the graph links a pair of vertices in V. A graph is *directed* if each edge is unidirectional. A graph is *weighted* if every edge is associated with a weight value. A graph is associated with a distributed program, which consists of a collection of LPs at the vertices with the messages that pass along the edges of the graph. Each vertex is bound to an LP and each edge denotes the relationship between a pair of LPs. The graph can represent a logical structure that is independent of the real structure of a distributed system, which can be used to reflect the properties of the underlying system. For example, the weight on each edge may denote the cost or delay in sending a message from one site to another site within the system.

A DIG-based program consists of a collection of LPs, which are built using the graph construct, and a main program. A graph construct consists of a directed conceptual graph (DGraph), an LP-to-vertex mapping, and an optional vertex-to-processor mapping. A graph construct is specified as follows:

- A *conceptual graph*: in which vertices represent the LPs of a distributed program and edges represent the relationships between LPs.
- A *LP-to-vertex mapping*: showing the binding of LPs to the vertices.
- An optional *vertex-to-processor mapping*: specifying the mapping of the logical graph to the processors of the underlying system. If this mapping is omitted, the run-time system will automatically decide the mapping.

Using the graph construct that is defined above, an application programmer can create an instance of a graph construct using the following steps:

Step 1: DGraph template declaration and instantiation.

A DGraph is a template for a logical graph that is defined in the model. It defines a construct as a conceptual graph describing the logical relationships between LPs, and it instantiates a graph instance and associates a name with the instance. The structure of a DGraph is a general type of logical graph, which is described as a list of vertices connected with edges. The language description (given below) defines the structure of a DGraph in Backus–Naur Form.

<DGraph-template> ::= DGraph *DGraph-name* '=' {{*<set-of-vertices>*}, {*<list-of-edges>*}}

<set-of-vertices> ::= *<range-of-vertices>* | *<vertex-list>*

<range-of-vertices> ::= *<vertex_no>*..*<vertex_no>*

<vertex-list> ::= *<vertex-list>*, *<vertex_no>* | *<vertex_no>*

<list-of-edges> ::= *<list-of-edges>*, {*vertex_no*, *vertex_no*}|*e*

A DGraph is the type identifier denoting the definition of a graph construct. The *DGraph-name* is an identifier of a graph construct. The *vertex_no* is an integer identifier of a vertex.

Step 2: DIG mapping.

Map LPs to the conceptual vertices of a graph and the vertices to the underlying processors. The LP-to-vertex mapping is defined as a set of (*vertex_no*, *LP_no*) pairs.

$$<LV\text{-}mapping> \ ::= \ \mathsf{LVMAP}\ LV\text{-}map\text{-}name\ `=\text{'}\ \{<vertex\text{-}lp\text{-}pair>\}$$
$$<vertex\text{-}lp\text{-}pair> \ ::= \ <vertex\text{-}lp\text{-}pair>,\ \{<vertex\_no>,\ <LP\_no>\}|e$$

LVMAP is the type identifier of an LP-to-vertex mapping. *LV-map-name* is the name of a mapping instance. The LP named in *LP_no* is mapped to the vertex identified by *vertex_no*.

The vertex-to-processor mapping is optional in the graph construct, which is specified by a set of (vertex_no, processor_no) pairs, in a similar form to the LP-to-vertex mapping.

$$<VP\text{-}mapping> \ ::= \ \mathsf{VPMAP}\ VP\text{-}map\text{-}name\ `=\text{'}\ \{<vertex\text{-}processor\text{-}pair>\}$$
$$<vertex\text{-}processor\text{-}pair> ::= <vertex\text{-}processor\text{-}pair>,\ \{<vertex\_no>,\ <processor\_no>\}|e$$

VPMAP is the type identifier of a vertex-to-processor mapping. *VP-map-name* is the name of a mapping instance. The vertex named in *vertex_no* is mapped to the processor identified by *processor_no*.

Step 3: DIG construct binding.

Given the declaration of a graph construct and its mappings, a graph instance can be created by binding the mappings to the graph. The CreateDIG function is used for this purpose.

CreateDIG
(*DGraph-name*, *LV-map-name*, *VP-map-name*, *DIG-instance-name*);

DIG-instance-name is the identifier of a file that specifies all of the information about the newly created instance. This information is useful in establishing the operating context for the LPs. *LV-map-name* is the name of an LP-to-vertex mapping and *VP-map-name* is the name of a vertex-to-processor mapping. The local operating context is created by

SetUpLocalDIG (*DGraph-name*, *DIG-instance-name*, *myvid*);

The *myvid* argument is the identifier of an instance of an LP. As a result of a call to the SetUPLocalDIG function, a local representation of the distributed instance of a conceptual graph is created and the information required by the communication protocols is generated.

Programming based on a graph-oriented model includes creating the graph construct and writing program code for the LPs using the *graph primitives*. A programmer can first define a graph construct that specifies the structure of a distributed program and then write code for the LPs. The programmer has enough flexibility to exploit the semantics of the graph construct to deal with different aspects of distributed programming in an identical way. In this way, the programmer is saved from the burden of writing dedicated program codes for implementing message passing and task mapping,

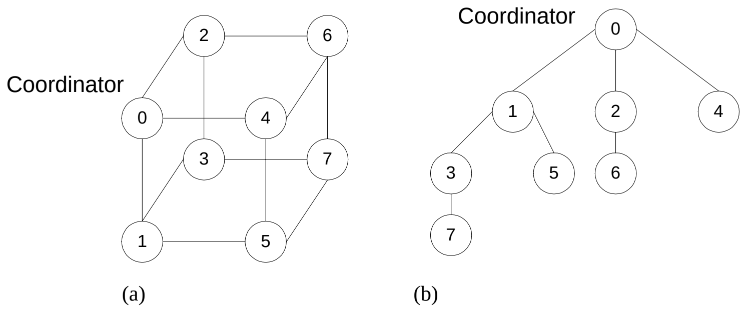

Fig. 1. A graph of a hypercube and its spanning tree.

and he/she can concentrate on designing the structure and the logic of the distributed program instead.

The following example explains the basic approach for using our graph-oriented distributed programming model. Our example is the computation of a global sum that is based on a hypercube, as shown in Fig. 1(a).

The hypercube is specified as the graph construct that is given below:

(1) *Conceptual graph*:

```
DGraph hypercube = { {0..7},
                     {{0,1},{0,2},{0,4},{1,3},{1,5},{2,3},{2,6},
                     {3,7},{4,5},{4,6},{5,7},{6,7}} };
```

This declaration defines a graph construct named hypercube. The first component of the definition, {0...7}, specifies all vertices in the graph. The second component specifies all edges, where each edge is defined as a pair of adjacent vertices.

(2) *LP-to-vertex mapping*:

```
LVMAP LVM = { {0,"Coordinator"}, {1,"Participant"},
              {2,"Participant"}, {3,"Participant"},
              {4,"Participant"}, {5,"Participant"},
              {6,"Participant"}, {7,"Participant"}} };
```

This declaration specifies a mapping that is named LVM, which maps between the vertices and the LPs. In our example, there are two types of LPs: the Coordinator receives and distributes the global sum, and the Participant calculates and submits the partial sums for a vertex and then collects the final global sum from the Coordinator. One approach to compute the global sum is based on deriving a spanning tree from the hypercube, as shown in Fig. 1(b), and then performing the calculation based on that tree [7]. This is an efficient algorithm because the number of messages that are sent between the processes is the minimum. The following text describes the program codes of the Coordinator and Participant.

```
Coordinator()
{
      float global_sum;
      MESSAGE message, re_message;
      int myvid;

      SetUpLocalDIG(hypercube, "GlobalSum", myvid);
      SpanningTree(hypercube, myvid, hypercube_st);
      PrepareMessage(message, myvid, "GLOBALSUM", No_Data);
      SendToChildren(hypercube_st, message);
      global_sum = my_value;
      while (ReceiveFromChildren(hypercube_st, re_message, "PARTIALSUM"))
        global_sum = global_sum + re_message.data;
      PrepareMessage(message, myvid, "RESULT_GLOBALSUM", global_sum);
      BroadcastMessage(hypercube_st, message);
}

Participant ()
{
      float partial_sum, global_sum;
      MESSAGE message, re_message;
      int myvid;

      SetUpLocalDIG(hypercube, "GlobalSum", myvid);
      partial_sum = my_value;
      ReceiveMessage(hypercube_st, re_message, "GLOBALSUM");
      If ( !IsLeaf(hypercube_st, myvid) ) {
        PrepareMessage(message, myvid, "GLOBALSUM", No_Data);
        SendToChildren(hypercube_st, message);
        While (ReceiveFromChildren(hypercube_st, re_message, "PARTIALSUM"))
          partial_sum = partial_sum + re_message.data;
      }
      PrepareMessage(message, myvid, "PARTIALSUM", partial_sum);
      SendToParent(hypercube_st, message);
      ReceiveMessage(hypercube_st, re_message, "RESULT_GLOBALSUM");
      global_sum = re_message.data;
}
```

The spanning tree can be derived automatically using the SpanningTree() primitive in the Coordinator, which is then implemented by the underlying support system of the DIG model. Each LP starts by creating a local operating context for its computation.

This is done by calling the primitive SetUpLocalDIG(*DGraph-name*, *DIG-instance-name*, *myvid*). Each LP has a unique identifier called myvid. The myvid identifier is bound to a processor ID in the *vertex-to-processor* mapping. The PrepareMessage() primitive is used to compose a message by assigning it the ID of the sender, a message type and some other parameters. A call to SendToChildren() with a given message causes that message to be sent over every edge emanating from a given vertex. ReceiveFromChildren() collects the messages from all edges emanating from a given vertex; one message is collected per edge.

The vertex-to-processor mapping, VPM, can also be specified in the graph construct. This mapping is optional. However, if it is missing, a default mapping will be generated. Given the conceptual graph and the LP-to-vertex mapping, the function CreateDIG(hypercube, LVM, "GlobalSum") can be called to combine the graph declaration and the mapping and produce an instance, called GlobalSum, of the graph construct.

A rich set of primitives is provided for performing various operations on the graphs. The operations on the graphs can be categorized into several classes [8], including communication among the vertices, subgraph generation, graph update and query. Using the graph-oriented model, a programmer can simply invoke such operations and base the calculation on the resulting subgraph and use of the primitives (such as SendToChildren, ReceiveFromChildren, SendToParent, and ReceiveFromParent).

The program code that is shown here is based on the C language. However, the programming language, which is used to encode the graph construct, is not important. Our model is applicable to other programming languages.

The realization of a graph-oriented model depends on the DIG, which provides a logical abstraction of the underlying operations. Also, the following functions need to be implemented:

Distribution of a graph: This is the representation of a graph in a distributed environment. A graph can be either directed or undirected. The distribution of a graph can be implemented on the processors in three ways: centralized, partitioned, and replicated. Implementations of the operations on the graph will vary for different forms of distribution of the graph.

Mapping: This is the method for mapping a graph to the underlying processors of a network. If a user specifies the mapping, the solution is straightforward. Otherwise, the run-time system will need to explore task-scheduling techniques to map the graph to the processors so that it makes efficient use of resources and/or speeds up the computations of the system.

Operations on a graph: These are the required operations on a graph and the distributed implementation of those operations needs to be designed.

Our graph-oriented programming model supports message passing and other useful inter-process communication (IPC) abstractions (such as *ports*, *binding* and *group communication*) that are used in many existing systems. In addition to the graph-oriented communications, the model also allows a programmer to exploit the graph construct to deal with other aspects of distributed computing. For example, a graph can be modified dynamically to reflect the reconfiguration of the system. In the rest of this paper, we focus on the properties and operations in our model for the dynamic configuration of a system.

4. Dynamic configuration management in the graph-oriented model

4.1. Dynamic configuration model

With DIG, a programmer can construct a distributed program using a logical graph. Both the structure of the graph and the structure of the underlying distributed system can be changed dynamically. Furthermore, the mapping of LPs to physical processors can also be changed dynamically. In this section, we describe the model in DIG for dynamic configuration. A dynamic configuration model specifies the methodology for dynamic reconfiguration and the required properties of the system components and the configuration system. The dynamic configuration model is actually built on the graph-oriented programming model and it provides special primitives and functions for configuration.

To achieve dynamic reconfiguration, the design of software components, the programming language and the supporting environment must possess special properties [19,23]. The term *software component* refers to a part of the software of a distributed system. A *component type* is the program code that is executed by a *component instance* [23]. Many component instances may be created from a particular component type. A component encapsulates its local data and functions, and it can communicate with other components through well-defined interfaces. In our configuration model, the instances of software components correspond to the LPs that are described as part of the graph-oriented programming model. The components can be coded in a conventional programming language such as C. The DIG system is implemented as a set of library routines for a conventional programming language. An LP has a format that consists of the statements of the language and additional DIG functions. The invocation of a function that is implemented by an LP is done by passing messages that contain the specified function name.

The systems that can accommodate reconfiguration usually have the properties of modularity and context independence. *Context independence* means that all interactions of a software component with its environment are explicitly described in the interface of that component [18]. In particular, the information about the configuration should be separated from the components. A software component is required to be context independent to ensure uniformity in its behavior in different configurations. This separation of the configurative information from the components of the system provides flexibility for the software reconfiguration being manipulated at run-time, thereby minimizing the influence on the components' uniformity. Therefore, when a change in the configuration occurs, the components do not need to stop execution, and only the separate configurative information has to be updated. In the following text, this configurative information is referred to as *configuration specification*.

The components should support modularity of the software. That is, the components can be written and compiled independently. All of the statements in the program code within a component should only refer to local objects and not refer to any shared objects, since such a reference would limit the flexibility of adding and removing components. The direct naming of component modules and communication entities must be avoided in any component, since it would limit the flexibility of the reconfiguration.

Instead, the names of components and communication entities should be specified in the configuration specification. Otherwise, when reconfiguration is needed, a programmer would have to stop an application and then edit the program code and compile it, thus making dynamic reconfiguration impossible.

A configuration specification is usually written in a special language called the *configuration language*, which is also used to write the specification for a reconfiguration of a system. This language should also allow the specification of unplanned reconfigurations to be made while the system is executing.

The graph-oriented distributed programs exhibit the properties that are described above. Based on the graph-oriented model, our approach to dynamic reconfiguration is to incorporate the configuration specification into the program design, eliminating the dependency of the code on the configurative information. Both the software components and their configuration are specified using a conventional programming language that is extended with library primitives. The configuration specification contains the definition of the context, a description of the interconnections among LPs and the optional allocation of the LPs to processors. Reconfiguration is also described using the same language. The modularity of the specification [19] is achieved by the logical graph construct. The communication and synchronization between the LPs are programmed in terms of the logical graph (using functions such as SendToChildren, ReceiveFromChildren, SendToParent, and ReceiveFromParent) rather than by indicating the explicit identifiers of the LPs. For dynamic reconfiguration, this approach allows the IPC interface (each provided at a particular vertex) to be decoupled from the actual LPs that respond to the messages. Each LP at a vertex refers internally to its peers during communication using only local names that will remain unchanged under different configurations of the system. Furthermore, the mapping of a logical vertex to a physical processor is performed by the underlying support system and the mapping is transparent to an application.

Dynamic configuration occurs for two main reasons: *operational* and *evolutionary* requirements [23]. The requirement for operational changes, such as dynamic expansion of worker processes (e.g., in a tree structure) in a master–slave paradigm, can be programmed into a distributed program because this requirement is indicated at the time the program is designed (see the first example in Section 5.2). However, evolutionary changes, such as expanding or adding new functionality or handling load imbalance and faults, cannot be predicted when a program is developed. The main difference between the two requirements lies in whether the configuration specification of a distributed program has to be modified at run-time to reflect the reconfiguration of the system. For the operational requirement, the configuration specification remains unchanged, whereas for the evolutionary requirement, the operations for configuration must be applied to the original specification to change its contents and also applied to change the distributed program.

Our model supports both operational and evolutionary changes. We have anticipated an incremental dynamic configuration process that can deal with an arbitrary unpredicted modification and an extension to the functionality without rebuilding the entire system. In DIG, a reconfiguration can be specified as a structural modification to the system by specifying the replacement, addition and reconnection of components. For example, we could expand the size of the hypercube dynamically by adding more vertices to the logical graph and by binding appropriate instances of the LPs to the vertices. The following primitives are provided for dynamic configuration:

1. int AddVertex(int vertex_no, char *graph_name); // add a vertex to a graph.
2. int DelVertex(int Vertex_no, char *graph_name); // delete a vertex from a graph.
3. int AddEdge(int start_vertex, int end_vertex, char *graph_name); // add an edge between two vertices in a graph.
4. int DelEdge(int start_vertex, int end_vertex, char *graph_name); // delete the edge between two vertices from a graph.
5. int MapLptoVertex(int LP_no, int vertex_no, char *graph_name); // map a local program to a vertex.
6. int UnMapLptoVertex(int LP_no, int vertex_no, char *graph_name); // remove the mapping of a local program to a vertex in a graph.
7. int MapVertextoHost(int vertex_no, char *hostname); // map a vertex to a processor.
8. int UnMapVertextoHost(int vertex_no, char *hostname); // remove the mapping of a vertex to a host.

In evolutionary changes, the sequence of operations to change a system can be applied progressively. Accordingly, the state of the configuration has to be considered and modeled. The operations must be specified in the context of the configuration states, which can be maintained in a (configuration) database by the dynamic reconfiguration manager.

4.2. A framework for dynamic configuration management

Because of the varied relationships among the interactions of the components and the requirements for consistency, a dynamic configuration management system is needed to support the required dynamic changes to distributed applications and systems. Fig. 2 shows our framework for dynamic reconfiguration. The shaded area shows the workflow of the dynamic reconfiguration on a distributed system. The requests for reconfiguration are sent to the validation module, which checks the validity of the requests. The validation module needs to inquire about the configurative information that is maintained in the configuration specification database. If the sequence of requests is valid, the reconfiguration module will update the corresponding graph construct and the configuration specification.

An essential component of dynamic configuration management is *dynamic reconfiguration manager*, which is responsible for validating a reconfiguration plan and for coordinating the distributed processes to execute the plan. The manager maintains a database of the configurative information associated with the distributed programs. Initially, when a distributed program is constructed, the dynamic reconfiguration manager reads and checks the content of the configuration specification and then builds the graph construct accordingly. Multiple instances of a graph construct can coexist where each one is associated with its own configuration specification. During the execution of a distributed program, the manager is ready to accept requests for reconfiguration that are specified in a reconfiguration plan and then validate and process the plan.

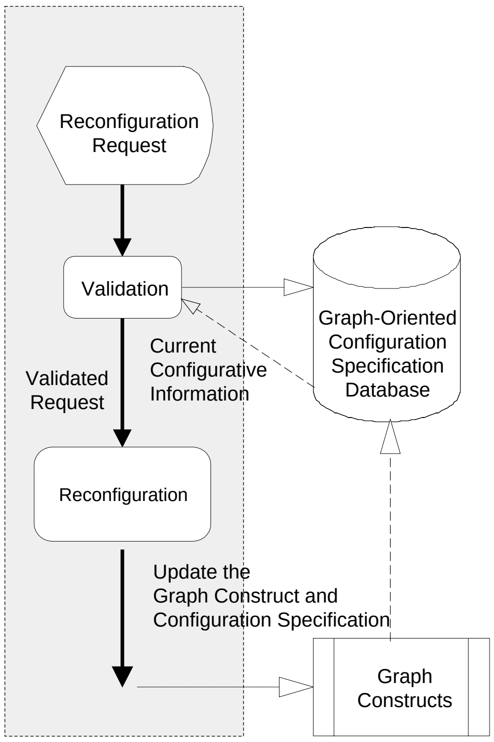
Reconfiguration Request
Validation
Validated Request
Current Configurative Information
Graph-Oriented Configuration Specification Database
Reconfiguration
Update the Graph Construct and Configuration Specification
Graph Constructs

Fig. 2. The dynamic reconfiguration framework.

Fig. 3 shows the component modules of the dynamic reconfiguration manager. The communication module is responsible for the interactions between the initiating program component and the LPs. It receives the requests for reconfiguration and returns the processing status to the appropriate module of the calling program after the reconfiguration has finished. The configuration specification module reads the configurative information from the configuration database and updates the configuration specification in the database according to the reconfiguration that was performed. The validation module performs the validation operations for the requests, as described in Fig. 2. This validation includes checking for the existence of a vertex or an edge that is planned to be deleted, maintaining a causal ordering among the requested changes, etc. The following primitives have been provided for validation:

1. boolean IsGraphExist(char *graph_name); // check whether a specified graph exists
2. boolean IsVertexExist(int vertex_no, char *graph_name); // check whether a vertex exists in a graph
3. boolean IsMaxVertexExist(char *graph_name); // check whether a maximum number of vertices already exist in a graph.
4. boolean IsEdgeExist(int start_vertex, int end_vertex, char *graph_name); // check whether an edge exists in a graph.

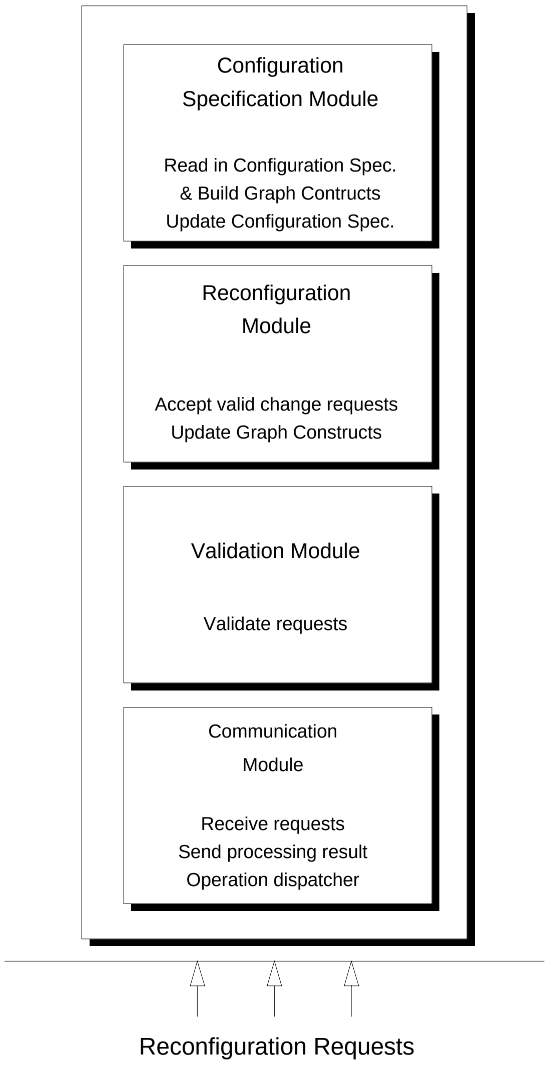

Fig. 3. Component modules of the dynamic reconfiguration manager.

5. boolean IsLPtoVertexExist(int LP_no, int vertex_no, char *dgraph_instance_name); // check whether a LP to vertex mapping exists in a graph instance.
6. boolean IsInsExist(int LP_no, char *dgraph_instance_name); // check whether a LP instance exists in a graph instance.
7. boolean IsInsMapped(int LP_no, char *dgraph_instance_name); // check whether a LP instance is mapped to a vertex in a graph.
8. boolean IsMaxInsExist(char *dgraph_instance_name); // check whether a maximum number of LP instances already exist in a graph.

9. boolean IsVertexValid(int vertex_no); // check whether a vertex has a valid ID ranging from zero to a maximum number.
10. boolean IsInsValid(int LP_no); // check whether a LP instance has a valid ID ranging from zero to a maximum number.
11. boolean IsVertexMapped(int vertex_no, char *dgraph_instance_name); // check whether a vertex has been mapped to a processor.

After the validation, the reconfiguration module executes the valid reconfiguration and updates the graph construct. An important requirement of the reconfiguration management process is to ensure that a dynamic change is performed in a safe manner to maintain a consistent state for the modified system. Standard consistency constraints, which are independent of the application, can be built into the validation module of the reconfiguration manager. However, it is believed, in general, that the consistency cannot be preserved without the support of the application programs that form the base-level components [17]. Therefore, the consistency constraints cannot simply be added as a management facility to an existing distributed system. We will not discuss this issue further in this paper.

5. A prototypical implementation

We have implemented a prototype of the graph-oriented distributed programming model [8]. This implementation is on top of a parallel virtual machine (PVM) [12] over a network of SUN workstations. The graph construct is provided to the programmers as a library of C routines. A distributed representation of a directed graph is used [9]. Each graph is assigned an identifier that is unique within the entire system, which is used to invoke the operations on the graph. Multiple graphs can coexist in the system and be run in parallel. DIG does not attempt to completely hide the message-passing paradigm at the level of program. A programmer still has a message-passing view of the system. However, this is an abstract view that is defined in terms of the edges in a logical graph. The programs that are based on the DIG model are bound to the vertices of a graph and integrated with the distributed graph operations and inter-vertex communications.

5.1. A central-server-based implementation

Based on the prototypical implementation of DIG, a prototype of the graph-oriented dynamic configuration management platform has been developed. A PVM is used as a virtual machine-configuration platform and as the communication library in the implementation of our example DIG model. A kernel of the DIG runtime system is run at each site on top of a PVM, as shown in Fig. 4. Programs can simply call DIG functions to interface with our platform using the daemons of a PVM.

A PVM provides a unified framework for developing parallel programs in an efficient and straightforward manner. It provides an abstraction for representing a collection of heterogeneous computer systems as a single virtual parallel machine. Moreover,

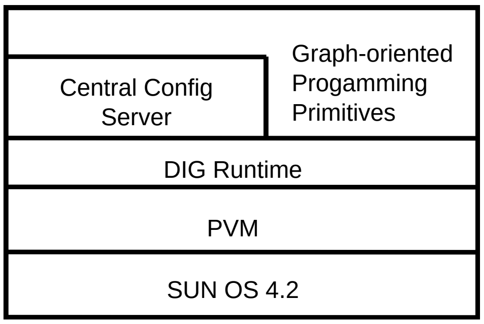

Fig. 4. The implementation architecture.

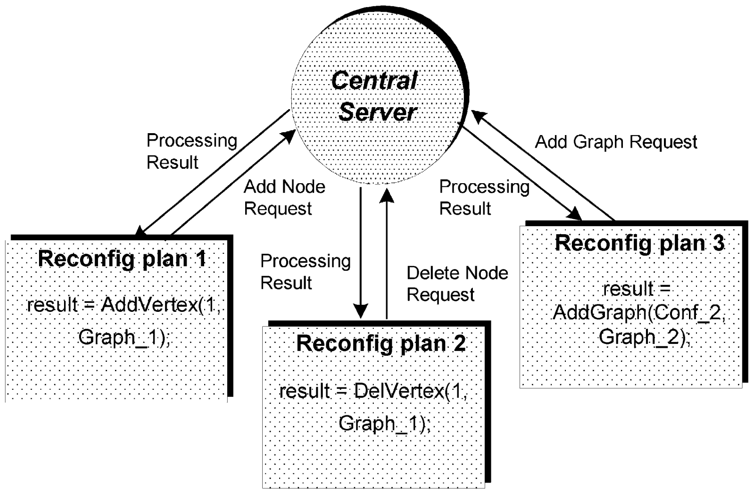

Fig. 5. Infrastructure of the dynamic reconfiguration management.

it handles message routing, data conversion and task scheduling across a network of heterogeneous computers. The user can write a collection of cooperative processes (called tasks) to access the resources of a PVM using a library of standard interface routines. The tasks on a PVM have arbitrary control that allows tasks to start or stop and it also allows computers to be added or deleted from the virtual machine at any time during the execution of an application. These properties make a PVM a useful tool for developing our graph-oriented distributed programming platform. In addition, with our platform being implemented on top of a PVM, it is also easy to port our platform to different hardware environments.

The prototype of the dynamic reconfiguration management process is implemented in a client-server infrastructure. Fig. 5 depicts this client-server infrastructure. The central server maintains the configuration specification of graph instances. The validation

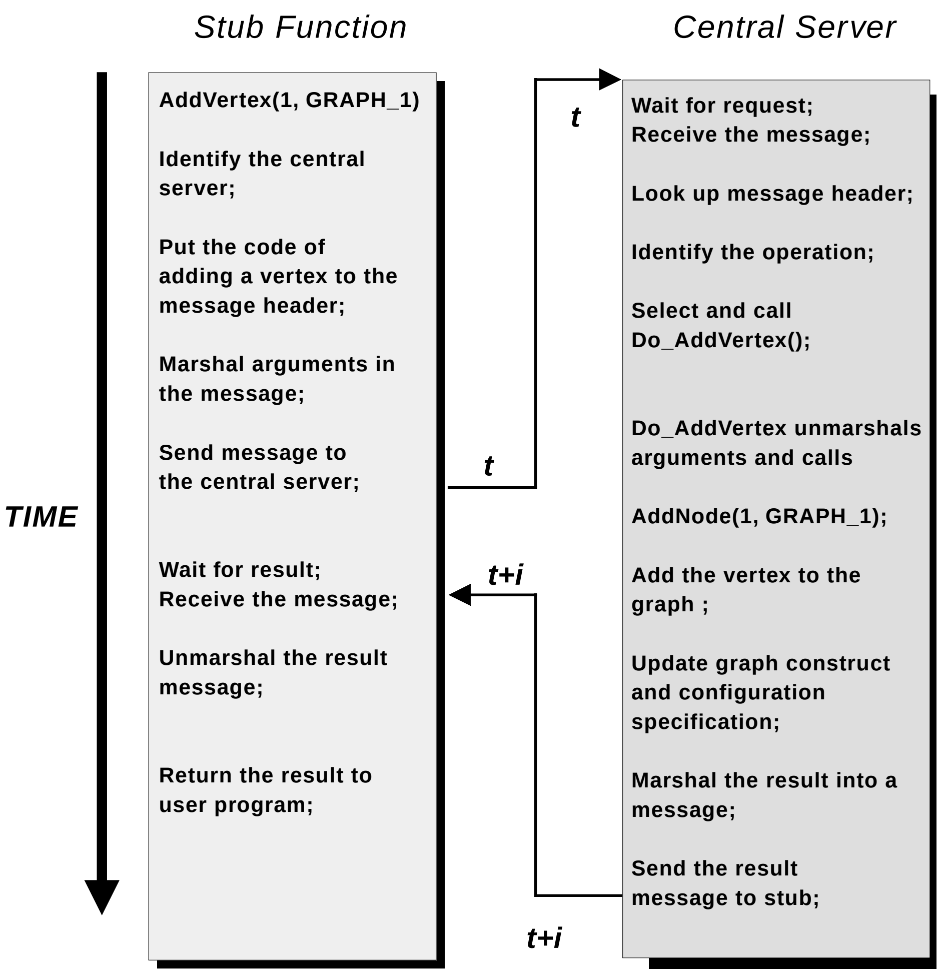
Stub Function
Central Server
TIME
AddVertex(1, GRAPH_1)
Identify the central server;
Put the code of adding a vertex to the message header;
Marshal arguments in the message;
Send message to the central server;
Wait for result; Receive the message;
Unmarshal the result message;
Return the result to user program;
t
t
t+i
t+i
Wait for request; Receive the message;
Look up message header;
Identify the operation;
Select and call Do_AddVertex();
Do_AddVertex unmarshals arguments and calls
AddNode(1, GRAPH_1);
Add the vertex to the graph ;
Update graph construct and configuration specification;
Marshal the result into a message;
Send the result message to stub;

Fig. 6. The interaction between a client's stub function and a central server.

and execution of a reconfiguration plan is performed as a client-server process, where the central server receives the requests from the program modules of a distributed application and initiates the reconfiguration.

The invocation of the reconfiguration functions on the central server is implemented using the primitives provided by a PVM. The central server blocks and waits for the incoming requests when it is in idle. When a reconfiguration is initiated, a function call is made to a stub function, which in turn invokes the management function at the central server. When a reconfiguration is completed, the server sends the result back to the stub function. Fig. 6 illustrates the interaction between the requesting program and the central server for a reconfiguration operation that involves adding a vertex to a graph.

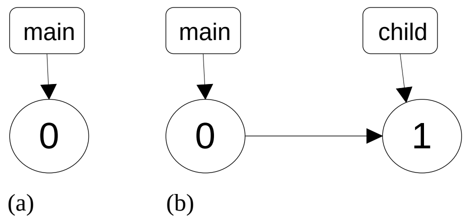

Fig. 7. The process of creating the binary tree.

5.2. Sample applications

The graph-oriented method for dynamic reconfiguration is powerful enough to model and implement various types of parallel and distributed programs. One of the popular parallel programming paradigms is called *tree computing*. In this paradigm, local processes in a parallel program are spawned dynamically and associated with a tree-like structure. The establishment of this tree structure results from the parent–child relationships between the processes. Such a tree-computing model is suitable for applications with an unpredictable workload (i.e., the total workload of a program is unknown before execution).

One sample application of tree computing is the *Split-Sort-Merge* algorithm. In this parallel sorting algorithm, one process contains a list of entities that are to be sorted and it spawns a second process and sends it half of the list. After the two sublists have been sorted separately, the first process is then responsible for merging the two sublists to produce the final sorted list. This split and merge process can be carried out recursively at different component processes. The spawning procedure continues until a tree with an appropriate depth is formed. Then, each process sorts its own sublist and a merge phase follows in which the sorted sublists are transmitted upwards along the tree branches with the intermediate merges being done at each appropriate node.

In this example, the main program is responsible for calling the primitives for graph-oriented reconfiguration to create an initial graph construct of the binary tree. Another type of local program is the child process, which may create vertices dynamically for the binary tree by sending requests for reconfiguration to the central server to change the configuration of the system during execution. The execution flow of the Split-Sort-Merge program is as follows.

- The main() program calls create_binary_tree("conf_spec", "binary_tree"), where conf_spec is the file containing the configuration specification and binary_tree is the name of the graph construct that is built using the information from conf_spec. Fig. 7(a) shows the initial graph that is described by conf_spec.
- As the main program executes, vertex 1 is added to the tree. An instance of the LP called child is added and mapped to vertex 1, which is then mapped to a host. An edge is added from vertex 0 to vertex 1. Now, a tree with two vertices has been created, as shown in Fig. 7(b).

```
# GRAPH NAME : binary_tree
[VERTEX]
0,1,2,3,4,5,6

[EDGE]
0,1;0,2;1,3;1,4;2,5;2,6

[LP]
0,main;1,child;2,child;3,child;4,child;5,child;6,child

[LPtoVERTEX]
0,0;1,1;2,2;3,3;4,4;5,5;6,6

[VERTEXtoHOST]
0,lisa;1,marge;2,homer;3,maggie;4,orion;5,mecury;6,aries
```

Fig. 8. Configuration specification for the binary tree.

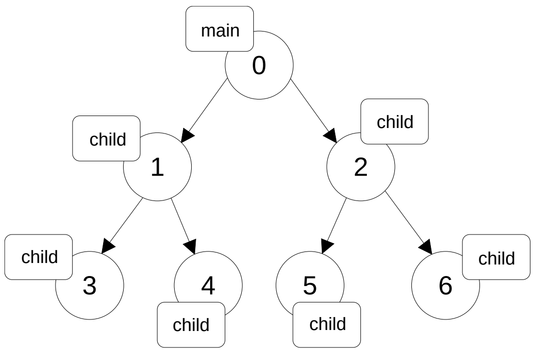

Fig. 9. The logical structure of the binary tree.

- The main program continues to add vertex 2. At the same time, the child that is bound to vertex 1 begins its execution, which will add two more vertices, 3 and 4, to the logical graph. The process continues until the list that is to be sorted at each vertex is short enough. In this example, a two-level binary tree composed of 7 nodes and 7 LPs is created. Fig. 8 shows the content of the final file for the configuration specification and Fig. 9 shows the logical structure of the binary tree.

This example shows an operational change in the configuration. The same principle is also suitable for performing an evolutionary change. Let us consider another example, the distributed optimization problem (DOP). The objective of a discrete optimization problem is to find an optimal solution from a finite or infinite (countable) set of solutions that satisfy the specified constraints. Problems in various fields can be categorized as DOPs, for example, planning and scheduling problems, the optimal layout of very large-scale integration chips, robot motion planning, test-pattern generation for digital circuits, logistics and control [21]. DOPs are usually solved by *search algorithms* that evaluate a set of candidate solutions to find one that satisfies the problem-specific criteria. In most DOPs, the solution set is so vast that it is not feasible to enumerate

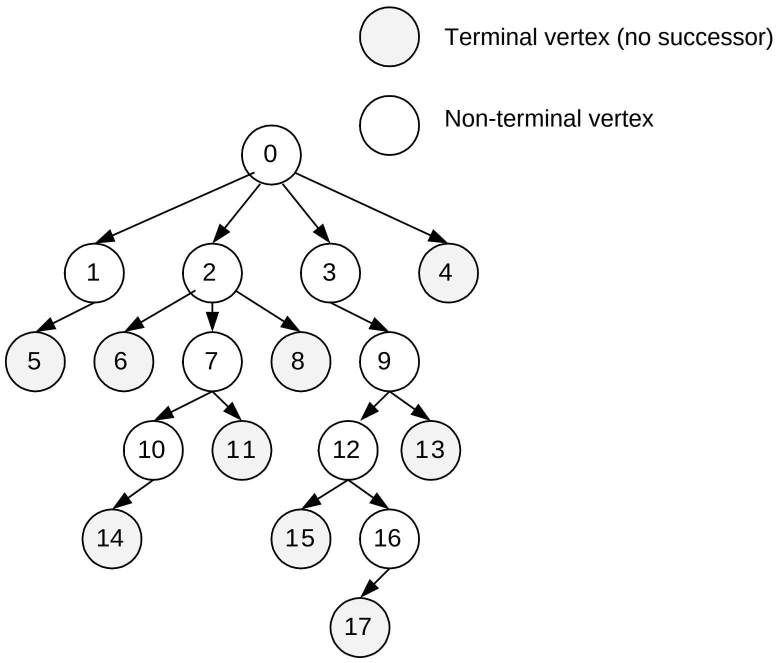

Fig. 10. The graph structure of a DOP.

through all candidates to determine the optimal solution. Thus, the solution space of a DOP problem can be represented as a tree. The DOP problem is solved by finding an optimal solution along a path in the tree. The tree can be expanded dynamically by generating new nodes at the ends of the existing branches to represent new candidate solutions during the search. Each new node that is generated leads to an unexplored part of the search space. The expansion and search of the solution space continues on the tree until an optimal solution has been found.

In the parallel search of DOPs, the search space is distributed onto multiple processes. Each process searches a subtree that can be expanded at run-time without any interaction with the other processes. Since the search space is generated dynamically, it is difficult to estimate the size of the search space before execution. Hence, it is impossible to partition the search space evenly among the processes in advance. DOPs can be represented in our DIG model. The dynamic reconfiguration of the model enables the evolution of the tree to reflect the dynamic expansion of the search space. This is an evolutionary change. Fig. 10 shows the graph construct (i.e., a tree) of a DOP. In the tree, the root (vertex 0) initially holds the whole search space. If the initial workload is high, the root node can generate new vertices to do parallel search on the subspaces. During the search, the workload on any vertex may become high due to the expansion of the search space. An overloaded vertex can spawn a new vertex to process part of the subspace. A new vertex is then bound to a new process that runs on an additional processor to perform the search on the subspace. The tree will not be expanded further when the workload on each vertex stays at a reasonable level (i.e., below a threshold). In Fig. 10, eventually 18 vertices are generated to perform the parallel search in the solution space.

In addition to the dynamic expansion of a graph, the workload can be scheduled among the processes dynamically. When a process runs out of work, it can also request extra work from another process that has an unexplored subspace. The requesting process submits a request for more work to the dynamic reconfiguration manager. The manager examines the status of all processes and transfers a portion of an unexplored subspace from an appropriate process to the requesting process. If an optimal solution is found or the whole solution space is exhausted, the computation terminates on all processes.

6. Conclusions

In this paper, we have described a graph-oriented model, called distributed implementation of graphs (DIG), for programming distributed applications. We have discussed dynamic configuration management in the proposed distributed programming environment. We have also described the design of a graph-oriented dynamic reconfiguration manager and presented a prototypical implementation on a parallel virtual machine. Examples of distributed programs have been constructed to illustrate implementations of the DIG model. We believe that the proposed framework provides a novel approach to programming reconfigurable distributed applications and systems. It is a powerful system for modeling the dynamic features of various kinds of distributed and parallel programs.

Some important issues have not been addressed in our work. For example, the ability to compose a system hierarchically from the component interfaces is the main underlying principle of the system configuration and reconfiguration. The recursive building of a binary tree in the split-sort-merge program demonstrates the hierarchical construction of a graph structure. The synchronization of the dynamic reconfiguration operations on a logical graph is essential to ensure the consistency of the application. The synchronized interaction between a client stub and a central server in the prototypical implementation demonstrates this capability of our model. Even so, a more comprehensive mechanism is required to support system-wide synchronization. In our future work, we are now investigating the issues associated with ensuring safe and consistent transitions of the states of the system during the reconfiguration. For example, determining the reconfiguration region, checking the states of the local programs (LPs) and transferring the states from an old version of an LP to a new one, and the transparent update of the configuration specification while the system continues execution.

Acknowledgements

This research was partially supported by a grant from the University Grant Council of Hong Kong. We would like to thank Mr. K.W. Yu for helping us in the implementation of the prototype.

References

[1] ADA, Reference Manual for the Ada Programming Language, Proposed Standard Document, USA Deparment of Defence, July 1980.

[2] G.R. Andrews, The distributed programming language SR-mechanisms, design and implementation, Software: Practice Experience 12 (8) (1982) 719–753.

[3] G. Andrews, R. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, G. Townsend, An overview of the SR language and implementation, ACM Trans. Programming Languages Systems 10 (1) (1988) 51–86.

[4] M. Barbacci, D. Doubleday, M. Gardner, R. Lichota, C. Weinstock, Durra: a task-level description language reference manual, Technical Report, 1991, available from http://www.sei.cmu.edu/publications/documents/91.reports/91.tr.018.html.

[5] T. Batista, N. Rodriguez, Dynamic reconfiguration of component-based applications, in: Proc. 5th Internat. Symp. on Software Engineering for Parallel and Distributed Systems, Limerick, Ireland, June 2000, pp. 32–39.

[6] T. Bloom, B. Day, Reconfiguration and module replacement in Argus: theory and practice, IEE Software Eng. J. 8 (2) (1993) 102–108.

[7] J. Cao, L. Fernando, K. Zhang, DIG: a graph-based construct for programming distributed systems, in: Proc. 2nd Int. Conf. on High Performance Computing, New Delhi, India, December 1995, pp. 417–422.

[8] J. Cao, L. Fernando, K. Zhang, Programming distributed systems based on graphs, in: M. Orgun, E. Ashcroft (Eds.), Intensional Programming I, World Scientific, Singapore, 1996, pp. 83–95.

[9] J. Cao, Y. Liu, L. Fang, L. Xie, Portable runtime support for graph-oriented parallel and distributed programming, in: H. Sudborough, B. Monien, D. Hsu (Eds.), Proc. Internat. Symp. on Architectures, Algorithms, and Networks (ISPAN'2000), IEEE Computer Society Press, Dallas, USA, December 7–9, 2000, pp. 72–77.

[10] N. Carriero, D. Gelernter, J. Leichter, Distributed data structures in Linda, in: Proc. 13th ACM Symp. on Principle of Programming Languages, St. Petersburg, FL, 1986, pp. 236–242.

[11] G. Coulson, G. Blair, M. Clarke, N. Parlavantzas, The design of a configurable and reconfigurable middleware platform, Distributed Comput. J. 15 (2) (2002) 109–126.

[12] Al Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM: Parallel Virtual Machine a User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, London, England, 1994.

[13] C.A.R. Hoare, Communicating sequential processes, Commu. ACM 21 (8) (1978) 666–677.

[14] C. Hofmeister, J. Purtilo, Dynamic reconfiguration in distributed systems: adapting software modules for replacement, in: Proc. 13th Internat. Conf. on Distributed Computing Systems, Pittsburgh, PA, USA, May 25–28, 1993, pp. 101–110.

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

[16] G.R.R. Justo, P.R.F. Cunha, Programming distributable system with configuration languages, in: Proc. Internat. Workshop on Configurable Distributable System, London, UK 1992, pp. 118–127.

[17] J. Kramer, Configuration programming—a framework for the development of distributed systems, in: Proc. IEEE Internat. Conf. on Computer Systems and Software Engineering (CompEuro '90), Israel, May, 1990, pp. 374–384.

[18] J. Kramer, Configurable distributed systems, editorial, in: Special Issue on Dynamic Reconfiguration, Software Engineering Journal, March 1993.

[19] J. Kramer, J. Magee, Dynamic configuration for distributed system, IEEE Trans. Software Eng. 11 (4) (1985) 424–436.

[20] J. Kramer, J. Magee, K. Ng, Graphical configuration programming, IEEE Comput. 22 (10) (1989) 53–65.

[21] V. Kumar, A. Grama, A. Gupta, G. Karypis, Chapter 8 Search algorithms for discrete optimization problems, in: Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin/Cummings, Menlo Park, CA, 1994, pp. 299–354.

[22] R. LeBlanc, A. Maccabe, The design of a program language based on connectivity networks, in: Proc. 4th Internat. Conf. Distributed Computing Systems, October 1982.

[23] J. Magee, Chapter 18 Configuration of distributed systems, in: M. Sloman (Ed.), Network and Distributed Systems Management, Addison-Wesley, Reading, MA, 1994, pp. 483–498.

[24] J. Magee, N. Dulay, J. Kramer, Structuring parallel and distributed programs, IEE Software Eng. J. 8 (2) (1993) 73–82.

[25] J. Magee, J. Kramer, M. Sloman, Constructing distributed system in Conic, IEEE Trans. Software Eng. 15 (6) (1989) 663–675.

[26] K. Moazami-Goudarzi, Consistency preserving dynamic reconfiguration of distributed systems, Ph.D, Thesis, Imperial College London, March 1999.

[27] P. Oreizy, R. Taylor, On the role of software architectures in runtime system reconfiguration, IEE Proc.-Software 145 (5) (1998) 137–145.

[28] G. Papadopoulos, F. Arbab, Configuration and dynamic reconfiguration of components using the coordination paradigm, Future Generation Comput. Systems 17 (8) (2001) 1023–1038.

[29] D. Peleg, Distributed data structures: a complexity-oriented view, in: J. Leeuwen, N. Santoro (Eds.), Proc. 4th Internat. Workshop on Distributed Algorithms, Lecture Notes in Computer Science, vol. 486, Springer, Berlin, 1990, pp. 71–89.

[30] N. Rodriguez, R. Ierusalimschy, R. Cerqueira, Dynamic configuration with CORBA components, in: R. Cole, R. Schlichting (Eds.), Proc. 4th Internat. Conf. on Configurable Distributed Systems, IEEE Computer Society Press, Silver Spring, MD, May 1998, pp. 27–34.

[31] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice-Hall, Englewood Cliffs, NJ, 1997.

[32] B.K. Totty, Experimental analysis of data management for distributed data structures, M.S. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.