Implementing configuration management policies for distributed applications

Distrib. Syst. Engng **3** (1996) 86–95. Printed in the UK

Implementing configuration management policies for distributed applications

Gerald Krause† and Martin Zimmermann‡

IBM European Networking Center, Heidelberg, Germany

Abstract. The central purpose of this paper is to present a novel framework supporting the specification and the implementation of configuration management policies for distributed applications. The introduced approach is part of a system called PRISMA (A platform for integrated construction and management of distributed applications). It provides an integrated application development environment comprising of a distributed application model, a specification language and several design, implementation and management tools. To reduce management complexity, we introduce a technique which permits the definition of configuration management policies as an integral part of an application specification. A configuration management policy is composed of one or more instantiation rules and termination rules for component creation and termination, and event-driven reconfiguration rules.

1. Introduction

The central purpose of this paper is to present a novel framework supporting specification and implementation of configuration management policies for distributed applications. The introduced approach is part of a system called PRISMA (a platform for integrated construction and management of distributed applications). It provides an integrated application development environment comprising a distributed application model, a specification language and several design, implementation and management tools.

Distributed applications can be characterized by the following features [14]:

- *Complexity*: typical distributed applications contain a large number of cooperating components and interconnections, resulting in a complex overall structure.
- *Intensive communication*: during runtime components use intensive communication and coarse-grained parallelism due to independent activities of the different components.
- *Dynamic changes*: a distributed application may be reconfigured during its lifetime due to evolutionary and operational changes, e.g. creation or deletion of components.

Long-running applications are a major challenge because they require specific mechanisms for reconfiguration. Changes should be executable during runtime, they should cause minimal disturbance to the running application and must be performed in such a way that leaves the running application in a consistent state. As a consequence, appropriate mechanisms for monitoring and controlling of components and interconnections are required, e.g. reconfiguration mechanisms which preserve consistency. An interactive management system is also needed which supports automatic control of distributed applications.

A precondition for configuration management is the availability of a specification technique for application configurations, i.e. specification of components (e.g. single components, component groups), communication relationships between components (e.g. one to many or many to many relationships), as well as specification of configuration constraints (e.g. the minimum cardinality of running component instances of a component type). To reduce management complexity, this must also involve a technique which permits the specification of configuration management policies as an integral part of an application specification.

The paper is organized as follows. Section 2 illustrates the basic characteristics of our application specification technique: specification of application configurations, components, interfaces and communication contexts. In sections 3, 4 and 5 we present a technique which permits the specification of configuration management policies as an integral part of an application specification. A policy is composed of one or more instantiation rules, termination rules and reconfiguration rules. Based on a rule-driven platform, in section 6 we illustrate a set of tools and the software architecture of our management system. Related work is discussed in section 7.

† Email address: krauseg@vnet.ibm.com
‡ Email address: mzimmermann@vnet.ibm.com

0967-1846/96/020086+10$19.50

```
clientServerApplication DISTRIBUTED APPLICATION
  COMPONENTS
        clients[ ]: Client {
                INSTANTIATION BY { user1, user2, ...}
        }
        documentServers[ ]: DocumentServer {
                INSTANTIATION REQUIRES ROLE Manager
                INITIAL MEMBERS n1
        workflowServer: WorkflowServer
        }
   BINDINGS
        FOR ALL c: clients
                c.docService -- ANY OF s IN documentServers
                c.wrkflService  -- workflowServer

         .....
```

Figure 1. Configuration specification.

2. Specification of distributed applications

A distributed application is composed of several cooperating components running on different physical nodes. From the application designer's viewpoint the basic building blocks of a distributed application are components with their interfaces and communication properties.

In the following, we illustrate the basic features of our specification technique. A client–server application is chosen to illustrate how our application model and the related specification technique can be used to specify the different aspects of a distributed application.

2.1. Application configuration

The configuration language is based on Darwin [7] but additionally provides configuration constraints, which describe configuration properties by predicates valid at all times during a running distributed application. A configuration specification describes the types of application components, the initial component instances, how these component instances are interconnected, and optionally a set of configuration constraints.

A component is either associated with a user and therefore called *interactive* or a software agent offering some service or executing some intermediate function. Besides 'single' component instances it is possible to define component groups whose size can vary during runtime. These groups are the basic concept for modelling dynamic user groups as interactive components in which new members join, and existing members leave the application at arbitrary times. Configuration constraints describe restrictions concerning the allowed configurations.

Figure 1 illustrates the specification of a client–server configuration. From the configuration point of view, the basic structure of a client–server application is that of an interactive client group communicating with one or more server components (document server and workflow server). The application configuration describes the initial client and server components (`COMPONENTS`), the topology (`BINDINGS`), and a set of configuration constraints (not shown in figure 1).

The component group concept is used to specify a collection of client and server components. Such a group can be annotated with an initial set of members and in the case of interactive components with users who are allowed to participate in the application, i.e. may create a client component instance (`INSTANTIATION BY`).

Regarding the bindings, a designer can specify explicit or implicit bindings. In the first case, a binding declaration contains concrete component instances, known at specification time. In the latter case (see figure 1), bindings are defined implicitly by component properties instead of certain component instances.

To express consistency requirements a distributed application can be enriched with constraints. Existence con-

```
INTERFACE DocServiceInterface
  CONSUMER INVOKES
      createDoc,
      searchDoc, ...

COMPONENT Client
  INTERFACES
    CONSUMER AT
      docService: DocServiceInterface
        ...
COMPONENT DocumentServer
  INTERFACES
    SUPPLIER AT
      docService: DocServiceInterface
    CONSUMER SUPPLIER AT
      replService: GroupInterface
```

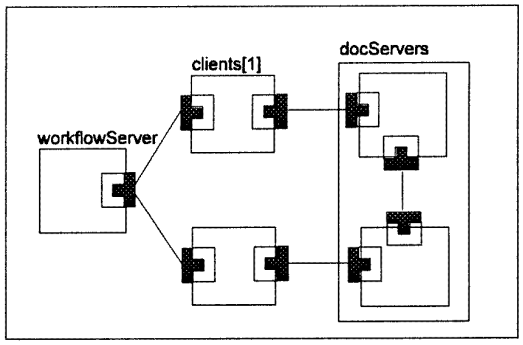

Figure 2. Component and interface specification.

straints define invariants in terms of required components and/or bindings. Placement of components onto a set of available computer nodes is determined by specification of placement constraints.

2.2. Components, interfaces and communication contexts

Components represent the distribution and configuration entities of a distributed application. A component type is described in terms of interfaces and communication contexts (see figure 2).

Interfaces are interaction points of components and describe the permissible interactions between cooperating components. Concerning the polarity, we distinguish between asymmetric interfaces which represent client–server behaviour and symmetric interfaces which enable a peer-to-peer cooperation. In addition to other approaches, the language allows the integration of a cooperation protocol. For each asymmetric interface of a component the polarity at the interface has to be determined, e.g. whether the component acts as a consumer and/or a supplier. The properties of each interface are described separately using the interface specification language. An interface specification consists of the operations which can be invoked and/or the operations which are offered. It is an optional part of an interface specification and enables the specification of regulations, i.e. we can specify which operation should be executed on which interface and by whom.

Communication contexts specify explicitly the communication requirements of distributed applications. For this purpose we provide a language which enables the specification of communication-oriented properties, such as the communication relation (connection-oriented or connectionless), the interaction type (message-oriented or operation-oriented) and the properties of a transport service [1, 2]. The latter are determined by the kind and amount of data exchanged between participants. For example, in interactive multimedia applications, the transport system requirements depend on the kind of media [17]. The communication requirements of a component are described by assigning one or more communication contexts to each interface.

3. Configuration management

Configuration management for distributed applications covers all phases during the lifetime of a distributed application: application specification, application creation and change management. In the specification phase, the initial application configuration, consisting of

- components
- bindings
- configuration constraints
- configuration management policies.

has to be defined using a configuration specification language (figure 3). After validation, creation of an initial application configuration can be initiated. This

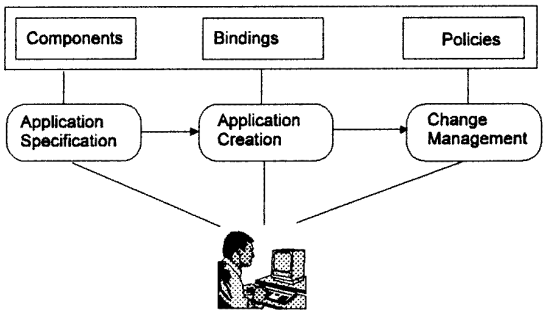

Figure 3. Management activities.

involves the creation of corresponding software processes and communication channels.

During runtime, change management is needed for the following activities:

- component management: creation, deletion and migration of components
- binding management: creation, deletion and modification of communication relationships between components
- status and history management: evaluation and modification of the application status, e.g. passivation of components
- policy management: definition of policies, modification and deletion of existing policies [16].

Figure 4 illustrates the basic infrastructure of our configuration management system. From the user's point of view a configuration management system provides an interactive graphical interface for specification of an initial configuration, for establishment of the corresponding processes and communication channels, for performing reconfiguration activities during runtime, and for observing the administrative and operational status of the application.

A management component must maintain an internal representation of a running distributed application in order to support operations for monitoring and control of a distributed application as well as for performing management policies. Such a computational representation consists of component representatives together with the communication relationships of each component instance, related configuration constraints, as well as configuration management policies.

4. Configuration management policies

According to the definition in [10, 11, 15], a policy is defined as a 'persistent specification of an objective to be achieved or a set of actions to be performed in the future or as an ongoing regular activity'. More formally, a policy is a pattern of the form 'condition $\rightarrow$ actions' (when? what? how?), where the condition specifies when something, namely the specified action, has to be done to preserve the policy.

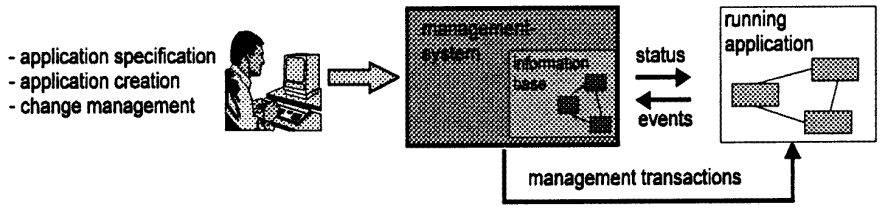

Figure 4. Management infrastructure for distributed application management.

4.1. Characteristics of configuration management policies

In our work, policies are used to influence the application configuration. They are the first step toward automated high-level management tasks currently maintained by human administrators.

Figure 5 illustrates the basic infrastructure of our policy-driven configuration management system. From the administrator's point of view there are operations for defining, querying and changing of policies. We distinguish two categories of reconfiguration activities. The first category contains management activities regarding creation of new components and termination of existing components. These types of activities are specified using the concept of *instantiation and termination rules* (IT rules). IT rules can be regarded as specific transactions which define the behaviour on creation and deletion of components. The idea behind the explicit specification of IT rules is similar to the concept of constructors and destructors as used in the programming language C++ to express instantiation and termination activities associated with object creation and deletion. Similarly to IT rules constructors and destructors are implicitly invoked each time a class object is allocated or deleted. The second category of management activities results from the fact that specific events from the application and network management, such as a crash of a computer node or the overload of a server component may lead to a reconfiguration activity. For this purpose, we provide a concept to specify a reconfiguration activity as a set of event driven *reconfiguration rules*.

The clear separation of policies into instantiation rules, termination rules, and reconfiguration rules not only supports modularity, reusability and extensibility of the management rule data base but also enables the designers to be guided more efficiently in building their policies. For instance, we can combine one set of IT rules with different sets of reconfiguration rules to provide application specific management policies. Additionally, the close relationship between reconfiguration rules and IT rules can be handled automatically by the management system. For example, whenever a component has to be created or terminated as part of a reconfiguration rule the corresponding instantiation and termination rules are triggered implicitly (see figure 5).

It should be noted, that during the runtime of a distributed application, it must be possible to establish new policies, and to modify or discard existing policies. Hereby new and changing management objectives can be achieved. For instance, there can be a set of policies to describe fault tolerant application configurations, and there can be another set to describe high performance configurations for stable and reliable system environments.

4.2. Instantiation and termination rules

During runtime of a distributed application, the basic configuration management activities are the creation and termination of application components. However, in most cases it is not sufficient simply to create and remove components. Instead there are additional initializing and terminating actions to be performed so that the resulting application configuration will be consistent. Consider for instance, a server group with replicated resources. The integration of a new server into such a server group implies that the existing group must be passive to prevent modifications of the resources during the join activity. Also, after creation it must be connected with the other servers of the group.

The concept of instantiation and termination rules facilitates a consistent integration of newly created application components into a running distributed application as well as a correct termination of components. They are described in terms of

- *administrative state*: preparation of a configuration change by passivating components or their interfaces ensuring minimal disturbance of components not involved in the change and preservation of the resulting configuration's consistency
- *management transactions*: sequences of basic management operations which are executed with transaction semantics. Depending on the application state, different management transactions may be executed.

A configuration change consists of several consecutive steps. In the case of terminating a component, the component must be set in a 'frozen' state where we can guarantee that all incoming interactions (at any of its interfaces) are completed, and no new interactions are initiated. This can be reached by passivating the component as a whole. In a second step, the current application state is examined and used to select an appropriate sequence of configuration change operations. This operation sequence is finally executed to perform the configuration change. To be able to restore the original configuration this is done within a transaction.

To illustrate this procedure, consider the termination of a server in the introduced example: Before removing it, all connected clients must be passive to prevent any

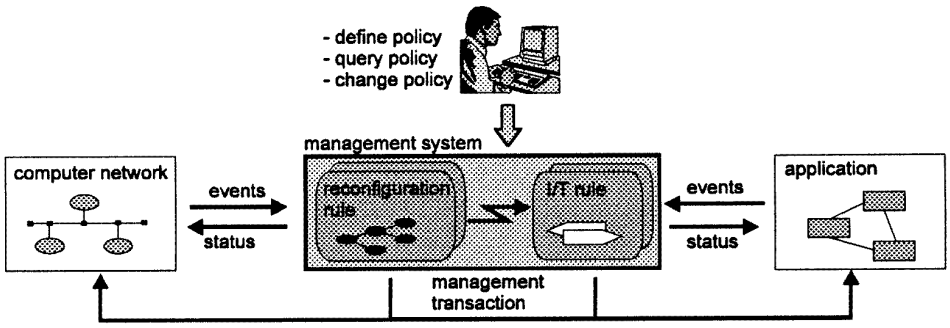

Figure 5. Integration of configuration management policies.

incomplete interaction. Then, as an optional terminating action to preserve the operability of the application, the client bindings must be delegated to another server. Finally the server component is removed.

In order to preserve consistency, the rules have to be executed according to a specific protocol, consisting of a sequence of phases. In the first phase, the administrative state of a component is set by the management system as specified in the rule. Before performing a rule the internal representation of the configuration is used to compute the resulting configuration by simulating the selected management transaction. If the new configuration does not satisfy the specified consistency constraints, the change will not be performed. Before executing the rules, checkpoints have to be set to be able to completely undo a change if some operation fails. After successful completion the checkpoints are cleared, and the remaining passive components are activated.

4.3. Reconfiguration rules

A reconfiguration rule describes a condition and a sequence of related management transactions to be executed when the condition is observed. A condition is a predicate expressed on the application model, i.e. configuration information about components, interfaces and roles. In contrast to instantiation and termination rules where a management transaction is selected by some application state, reconfiguration rules are triggered by events.

We distinguish between periodic events generated by a scheduler, and change events to reflect a state change of some application or network entity. The latter includes events from the network management (e.g. shutdown of a node) and/or application management (e.g. overload of a component). For example, depending on the current load of a server, a reconfiguration rule could describe that replication has to be initiated.

In order to get information from the underlying network, we extend the basic management architecture by an additional network management component (see figure 6). From the network management point of view, there are operations to indicate events relevant for the application management, e.g. the event `prepareForShutdown` indicating a shutdown of a computer node (figure 6). In the opposite direction the network management component provides operations to receive events from the application management

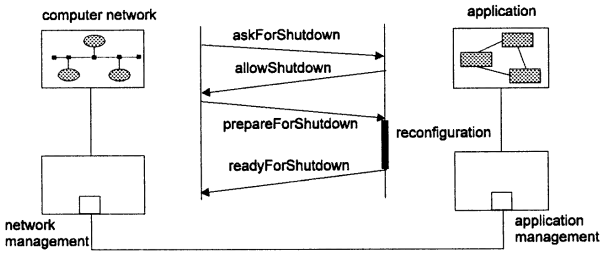

Figure 6. Interaction between network management and application management.

(e.g. the event `readyForShutdown` in figure 6) and operations to acquire resource properties of computer nodes. The latter operations are essential for determining alternative nodes for automatic placement at component creation time and migration of components during runtime. Application events are generated by notifications of application components. Performance monitoring events are of special interest, because they can result in the need to reconfigure the application. Therefore, metric objects are integrated into the application components. They are responsible for notifying a management component when relevant value changes take place.

A metric object consists of a metric element such as a counter or a gauge, and a description when to send an event notification [3]. Hereby, the filter mechanism is incorporated into the application component. When a component is prepared for the integration of metric objects, they can be created, removed, activated and passivated during runtime to avoid any unnecessary performance loss and to be able to adjust them to the minimum set of required events. In order to reduce the huge amount of event notifications to a reasonable size, an intermediate filter as a discriminator forwards relevant events and discards all other.

5. Policy specification

In this section we outline our specification language for instantiation and termination rules and for reconfiguration rules. Both languages are based on the fact that it can be foreseen what kind of actions have to be performed for what purpose, independent of any concrete configuration. Therefore the languages must provide

expressive capabilities to reference the building blocks of an application specification, i.e. names of components, contained interfaces, etc. Rather than describing the full flavour of both languages, the most important constructs are presented in the context of examples.

5.1. Instantiation and termination rule specification

According to the different aspects of a configuration change introduced in the previous section, a rule description is divided into multiple parts. In figure 7 an example for a termination rule for a document server component is shown; it consists of the parts `CONSTANTS`, `ADMINISTRATIVE STATE`, and `TRANSACTIONS`.

The `CONSTANTS` part is optional and can be used to define some constants which may be referenced throughout the rest of the rule. Here, the set constant `boundClients` designates all clients bound with the server to be terminated. With the constant `altServer` an alternative server is determined to which all the `boundClients` could be bound.

Next, in the `ADMINISTRATIVE STATE` part the required frozen state of the server is specified. Here, all bound clients must be passivated before the server can be terminated. Finally, the configuration change operations are determined. First, all the concerned clients are bound to the alternative server, and then the server can be terminated.

In more complex examples, another optional part is included in a rule where the management transaction is selected by querying the current application state. In this case, several named transactions are specified.

5.2. Reconfiguration rule specification

In order to explain the specification of reconfiguration rules, we use a migration rule for the document servers which should take place when a shutdown event notification is sent by the network management. Figure 8 shows the specification. The overall structure of the rule consists of a part containing the conditions to be observed, and an action part which describes the actions to be performed for satisfied conditions.

In the example above, the first clause of the `shutdown` condition is a notification `prepareForShutdown` for a specific node which is indicated by the network management (see figure 6). If this clause becomes true, the next clause checks all other conditions required for migration for all servers running on that node. First, a server on the affected node must be able to migrate (which is not necessarily true for any kind of components). Then, there also must be an alternative node to migrate to which fulfils the resource requirements of the server.

Now, whenever a `shutdown` condition becomes true, the corresponding actions are performed. For each server of the set computed in the condition, an appropriate node is selected for migration. Finally, the shutdown request will be confirmed by a `readyForShutdown` notification sent back to the network management system (see figure 6).

6. Tools and software architecture

For the implementation of a distributed application we have developed an object-oriented software architecture where components, interfaces and roles are realized by C++ classes. In the following, we concentrate on the implementation of the management system, especially the implementation of configuration information, IT rules and reconfiguration rules.

6.1. Architecture

The functionality which must be implemented for a configuration management application results from the underlying application model. A management component has five essential parts (see figure 9, right part):

- representation of configuration information
- representation of configuration management policies
- configuration management application
- communication infrastructure
- rule interpretation system

A computational representation of an application configuration and its parts is stored in an information base consisting of component representatives with their bindings and roles, related constraints, instantiation and termination rules and reconfiguration rules.

6.2. Representation of management information

The top-level object of the information base is a configuration object which serves as a local computational representation of a running distributed application. It is composed of component objects as local representatives of application components and is obtained from a configuration specification.

Each component representative has three essential parts [19]:

- *configuration information*: this consists of information about interfaces, roles and bindings of an application component as well as related type information. Operations are provided to access and manipulate the configuration information, e.g. creation of a new binding.
- *management functionality*: this part contains information about how to access the corresponding running application component. It includes a reference to a communication object to enable access to the remote managed application component.
- *graphical representation*: the graphical representation serves as the interface to the human administrator. It contains information for the representation of components, interfaces, roles etc as graphical icons and operations for access and manipulation of icons. For example, data to represent position and colour of component icons as well as methods to move and delete component icons.

```
clientServerApplication DISTRIBUTED APPLICATION
  COMPONENTS  ...
  BINDINGS    ...
  CONSTRAINTS ...

  TERMINATION RULE FOR documentServers
    CONSTANTS
      boundClients = { c IN clients WITH c -- documentServers [THIS] }
      altServer = ANY OF documentServers MINUS { documentServers [THIS] }

    ADMINISTRATIVE STATE
      FOR ALL c : boundClients (
        STOP c
        )
      STOP documentServers [THIS]

    TRANSACTIONS
      DELEGATE BINDINGS OF documentServers [THIS] TO altServer
      REMOVE documentServers [THIS]
```

Figure 7. Example of a termination rule.

```
clientServerApplication DISTRIBUTED APPLICATION
  COMPONENTS  ...
  BINDINGS    ...
  CONSTRAINTS ...

  RECONFIGURATION RULES
    CONDITIONS
      shutdown : prepareForShutdown (NODE n) AND
                 NOT EMPTY serverSet = { s : documentServers WITH (
                         s HAS LOCATION n AND
                         s HAS PROPERTY migration AND
                         NOT EMPTY nodeset = { m : NODES WITH (
                                 memory (m) >= memory_needs (s) AND
                                 adminState (m) = active ) } ) }
    TRANSACTIONS
      ON shutdown DO
        FOR ALL s1 : serverSet (
          SELECT n1 FROM nodeSet WITH
            memory (n1) = max (memory (nodeSet));
            MIGRATE s1 TO n1;
          )
          readyForShutdown (n);
```

Figure 8. Example of a reconfiguration rule.

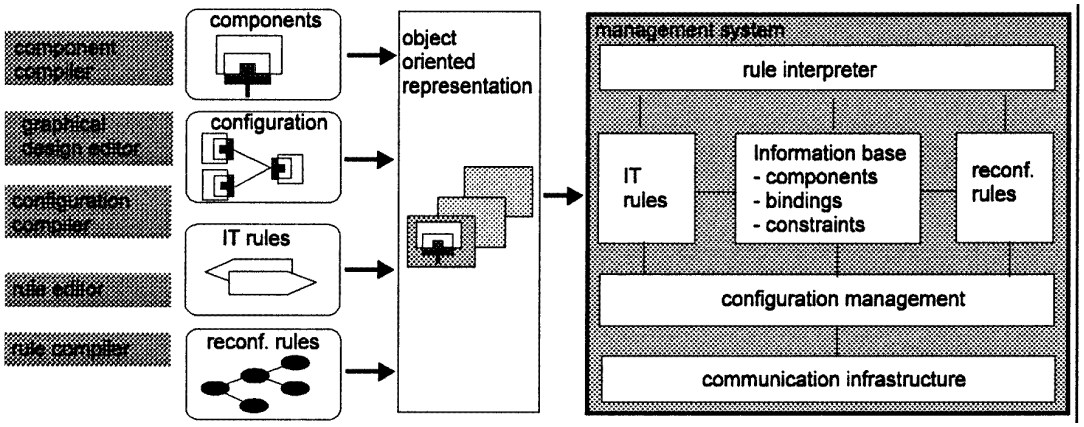

Figure 9. Architecture of a management component.

Using the object-oriented paradigm, we developed a set of classes for the basic elements of a configuration specification. These classes serve as a foundation for building an information base to enable access and modification of an application configuration [20]. A base class contains all the information about the currently valid component instances with their interfaces, roles and bindings and the location of the components. From this base class we derive a class which is used to represent a configuration associated with constraints. Constraints are mapped onto corresponding parse tree objects. These tree objects keep all the information required to perform consistency checks. Whenever a check of a constraint is initiated, the related tree is easily walked through.

In the existing prototype, the configuration information is based on a central file system. Future work will concentrate on an implementation using an object-oriented data base. This enables not only direct and natural handling of persistent C++ objects but also supports enhanced navigator mechanisms within the object store.

6.3. Representation and implementation of rules

The management application for execution of reconfiguration activities is mainly based on the integration of a rule interpretation system. We use Nexpert Object [12] as a platform for rule based systems. This approach permits us to use a general purpose rule interpreter. The representation of information and rules in Nexpert Object conforms to our application model. Especially, the expert system shell has an object oriented data model to express descriptive information. Dynamic knowledge is represented by situation–action statements stored as rules. The situation part of a rule consists of one or more conditions that must apply, if the action part is to be triggered during knowledge processing. This knowledge presentation allows us to represent the elements of a distributed application by objects in Nexpert Object.

Before processing of rules, in a first step, rules are transformed into the computational representation of Nexpert Object. In the second step, these rules are executed by the built-in rule interpreter. During knowledge processing Nexpert Object interacts with the configuration object, which is part of the information base. In the situation part of the rules information about the application configuration (components, interfaces, roles, bindings) is fetched using method calls of the configuration object and its contained objects. Similarly, in the action part of the Nexpert Object rules the configuration object is modified or extended using appropriate method calls.

A configuration management application is responsible for handling incoming events from the graphical user interface as well as from the remote application components.

6.4. Tools

To support the specification, implementation and management of distributed applications, we developed a set of tools in C++ (figure 9, left part). There are tools for

- specification of interfaces, components and application configurations (design editor, see figure 10)
- specification of configuration management policies (text editor)
- validation of a specification (interface compiler, component compiler, configuration compiler)
- generation of an internal object-oriented representation
- interactive graphical management system.

We use our configuration and management system for a distributed application from the Message Handling System (MHS) domain, consisting of user agents, message store components and message transfer agent components. However, the existing prototype does not support the interaction between application management and network management. Events from the network management must be notified to the application management by explicit user interactions.

Currently, we concentrate on configuration and management of telecooperation applications. Our telecooperation infrastructure is composed of document servers, workflow servers, and conference servers based on Lotus Notes as the document application platform and FlowMark as the workflow application platform.

The experiences gained so far showed that the management functions provided at component level are not sufficient to meet the requirements of complex distributed business applications. The existing configuration and management platform must be extended to enable more sophisticated techniques that support also management of fine-grained objects.

7. Related work

Rex [8] focuses on an integrated methodology and associated support tools for the development and management of parallel and distributed systems. A distributed application is structured as a set of software modules and communication ports. The structure of a distributed application is described by the separate configuration language Darwin which is based on experiences using the Conic configuration language. Darwin includes facilities for hierachic definition of composite objects, for parameterization of objects, for multiple instantiation of components and recursive definition of components. Components are objects and have well defined interfaces specified by an interface specification language. A component may be implemented in a range of heterogeneous programming languages.

Our approach can be regarded as an extension of the work done in Rex and Conic. At interface level we provide the concept of a cooperation protocol which can be used to describe the cooperation between application components as well as to specify the interaction between management and application components. In contrast to Rex, we use an explicit representation of communication properties and management interfaces as an integral part of a component specification. Our configuration language provides means to specify constraints, explicit and implicit bindings, and configuration management policies.

Polylith [4, 13] is a platform for development and management of distributed software applications. It provides similarly to Conic a language for describing the application structure as well as a software bus for managing the runtime activities. Based on an application specification, the software bus initiates the establishment of the distributed application, i.e. execution of the modules and creation of communication channels between modules. Insofar, Polylith differs from our approach in similar ways to Rex.

Meta [9] aims to provide a technique for integration of management functionality into application code. The basic idea is to instrument an application with sensors and actuators. A sensor represents part of the state of the monitored application, e.g. built-in sensors for

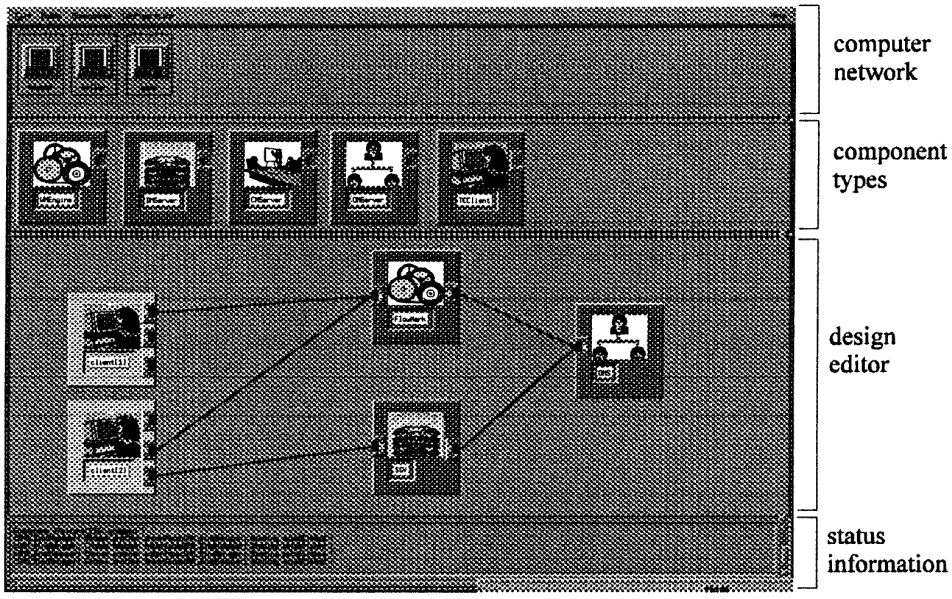

Figure 10. Graphical design editor.

obtaining statistics about utilization of memory and processor. Actuators provide operations for changing process priority, migrating processes to another machine or restarting a failed process. A control program observes the application behaviour through interrogating sensors which return values of the application state and its environment. Correspondingly, the behaviour of the application can be altered using the concept of actuators. On top of this functionality, a rule based language is provided which supports the specification of management policy rules. Each rule is composed of a condition part and a sequence of expressions involving actuators and sensors.

In contrast to Meta, our specification technique is used for describing application- and management properties of a distributed application. Moreover, based on our application model we support mechanisms to automatically determine the management properties of an application component. For example, the type of binding management required for an application component can be obtained by analyzing the binding properties of the application interfaces as part of the component specification.

The standardization activities in Open Distributed Processing (ODP) [5, 6] aim to provide a support environment for distributed applications. Based on the object-oriented paradigm, a distributed system is considered from different viewpoints, each of which is chosen to reflect one set of design goals. The resulting representation is an abstraction of a system; that is a specification which recognizes some distinctions and ignores others. The current work on ODP defines five viewpoints (concerning enterprise, information, computational, engineering, and technology aspects). However, the terminology of ODP has not been fully developed and is still evolving.

Our work can be regarded as part of the engineering and computational viewpoint. The computational model describes two major aspects. The first part contains an interaction model which introduces the concepts of invocation and announcement to represent different types of interactions. The second part is concerned with the configuration of objects. There is a close relationship between the terms introduced in this paper and those in the ODP reference model. Our concept of configuration constraints, instantiation and termination rules may serve as a specification technique for the corresponding ODP terms.

8. Conclusion

A new approach for specification and implementation of configuration management has been introduced. Consistent change management is supported using the concepts of constraints, instantiation rules, termination rules, and reconfiguration rules. The major contributions and extensions aim to provide an integrated construction and management methodology as well as methods to define reconfiguration activities as an integral part of an application specification.

The clear separation of policies into instantiation rules, termination rules, and reconfiguration rules has several advantages. First, it supports modularity, reusablity and extensibility of the management rule data base. Additionally, our approach supports more flexibility in combining different rules taking into account the specific application requirements. Forming an integral part of a distributed application specification, generic policies can be defined for different cooperation paradigms, e.g. for various types of client–server applications.

At implementation level, the different aspects are integrated in a general object-oriented architecture supporting modularity and reuse of software. To support the implementation and management of distributed applications, we developed a set of tools for the mapping of application specifications onto an object-oriented implementation model. It

includes tools for selection and initialization of managed objects based on a library of C++ managed object classes. A generic interactive management system enables establishing, monitoring and modifying of distributed applications as well as definition and modification of management policies. Future work will concentrate on the integration of performance and fault management. Moreover, the experiences gained from the prototype implementation will serve as a basis for implementation of decentralized management architectures [18], which are needed to manage larger applications.

References

[1] Berghoff J 1993 Development and management of communication software systems *Technical Report 6/93* J W Goethe University, Frankfurt/Main

[2] Feldhoffer M 1995 Model for flexible configuration of application-oriented communication services *Comput. Commun.* **18** 69–78

[3] Festor O and Zörntlein G 1993 Formal description of managed object behaviour—a rule based approach *3rd Int. Symp. on Integrated Network Management (San Francisco, 1993)* (Amsterdam: North-Holland) pp 45–58

[4] Hofmeister C and Purtilo J 1993 Dynamic reconfiguration in distributed systems: adapting software modules for replacement, *13th Int. Conf. on Distributed Computing Systems (Pittsburgh, 1993)*

[5] Information Technology–Basic Reference Model of Open Distributed Processing–Part 2 1993 Descriptive Model

[6] Information Technology–Basic Reference Model of Open Distributed Processing–Part 3 1993 Prescriptive Model,

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

[8] Magee J, Kramer J, Sliman M and Dulay N 1990 An overview of the REX software architecture *2nd IEEE Workshop on Future Trends of Distributed computing Systems in the 1990s (Cairo, 1990) IEEE Press* pp 396–402

[9] Marzullo K, Cooper R, Wood M and Birman K 1991 Tools for distributed applications management *Computer* August, 42–51

[10] Moffett J D and Sloman M S 1991 The representation of policies as system objects *SIGOIS Bull.* **12** 171–84

[11] Moffett J D and Sloman M S 1993 Policy hierarchies for distributed systems management *IEEE Select. Areas Commun.* **2** 1404–14

[12] *Nexpert Object Version 2.0* 1991 Functional Description Model (Neuron Data, Palo Alto)

[13] Purtilo J 1990 The polylith software toolbus *CDS Technical Report 2469* University of Maryland

[14] Schill A 1991 Distributed system and execution model for office environments *Comput. Commun.* **14** 478–88

[15] Sloman M 1994 Policy driven management for distributed systems *J. Network Syst. Management* **2** 333–60

[16] Sloman M 1995 IDSM & SysMan common architecture, Domain & Policy Concepts *SMDS'95 (Karlsruhe, 1995)*

[17] Steinmetz R and Meyer Th 1992 Modelling distributed multimedia applications *Int. Workshop on Advanced Communications and Applications for High Speed Networks (Munich, 1992)*

[18] Yemini Y, Goldszmidt G and Yemini S 1991 Network management by delegation *Integrated Network Management II* ed I Krishnan and W Zimmer (Amsterdam: North Holland) pp 95–107

[19] Zimmermann M and Drobnik O 1993 Specification and implementation of reconfigurable distributed applications *2nd Int. Workshop on Configurable Distributed Systems (Pittsburgh, 1993)*

[20] Zimmermann, M, Berghoff J, Dömel P and Patzke B 1994 Integration of managed objects into distributed applications *IFIP/IEEE Int. Workshop on Distributed Systems, Operations & Management (Tolouse, 1994)*