Chapter 3

Instantiating Descriptions of Organizational Structures

H. Edward Pattison, Daniel D. Corkill and Victor R. Lesser

Abstract

Instantiating and maintaining large distributed processing networks requires an explicit description of the system's organizational structure. Such a description identifies the system's functional components, their responsibilities and resource requirements, and the relations among them. Existing languages with features for describing organizational structure are inadequate for this task because they cannot describe the complex domain-specific relations found in many organizations. EFIGE is a language for specifying such relations. EFIGE aids the instantiation of these relations by allowing them to be constrained from the perspective of their members, and by allowing preferences to be expressed among instances of them. This chapter describes EFIGE and shows how relations with complex constraints may be implemented.

3.1 Introduction

The need to describe large and complex process structures—in order to instantiate them on specific processor configurations and to provide information to the operating system for resource allocation decisions and communication routing—has been recognized by a number of researchers, and they have developed languages for this purpose. These languages include DPL-82 [1], HISDL [2], ODL [3], PCL [4], PRONET [5], and TASK [6]. However, these languages are very weak in their ability to specify the complex processing structures necessary for the next generation of network architectures and distributed applications. This is especially true for applications with closely interacting tasks implemented on networks which are heterogeneous compositions of databases, effectors, sensors, and processors with various processing speeds and memory sizes. For example, the specification of the processing structure of a distributed processing network that performs signal interpretation requires a complex, domain-specific, communication relation between interpreting nodes and sensing nodes. This communication relation requires each interpreting node to communicate only with the smallest group of sensing nodes that can provide it with information about the region for which it is responsible. At the same time, each sensing node is required to communicate with a limited number of integrating nodes in order to minimize the time it must allocate for communication.

The specification of such complex process structures involves identifying functional components (such as interpreting and sensing nodes), their responsibilities (providing interpretations of the signals detected in a particular region) and resource requirements (processor speed and memory size, knowledge about interpreting signals, etc.), and the relations among them (communication and authority). Together, this information is a specification of the system's *organizational structure*. We see specification of organizational structure as not just parameter substitution and macro expansion, but rather a problem of *organizational planning* under conflicting instantiation constraints. These constraints arrive from the need to specify complex relations among the components of an organization. Relations include communication relations, authority relations that specify the importance given to directives from other nodes, and proximity relations that specify spatial positioning among objects. All of these relations may be complicated by interacting constraints. This was true of the communication relation between sensing and interpreting nodes given above, and is true of other relations as well. For example, a producer of a product whose value decreases with time may require that it be located near the consumer using the product or that both be located near nodes of a reliable transportation network.

Existing languages have implemented a few specific relations but their approach is limited. A communication relation, for instance, is described by explicitly stating that process X is to communicate with process Y. If the processes may be replicated, this statement becomes X_i communicates with Y_i, where i identifies a specific copy of each process. This form of description is not general enough. If Y_3, for example, is lost due to node failure, X_3 might as well be lost. Any information it was to have received from Y_3 will not be forthcoming and it will be idle; the production of any information it was to have sent Y_3 will consume system resources in vain. Since the description specifies only that X_3 is to communicate with Y_3, there is no way to find a substitute and one cannot be created because the characteristics of Y_3 that made communication with X_3 important are unknown.

Both the ability to specify more complex relations and to allow network designers to specify domain specific relations (such as the communication relation given above) are needed. Instead of requiring designers to specify communication relations as point-to-point connections, they should be asked to supply the criteria by which such pairings can be determined. The criteria that a member of one domain of a relation uses to recognize an acceptable member from another domain are called ***constraints***. Constraints ***specify*** a relation because they indicate which pairings of a member of one domain with the members of another are permissible. More precisely, a relation defines a subset of the ordered pairs (in general, n-tuples) that is the Cartesian cross-product of each domain of the relation, where each constraint in the relation is a predicate that selects some of the pairings as more significant than others. We will more loosely describe constaints as defining a new, more restricted relation, by ***refining*** the definition of a more general relation.

In this chapter we describe a language, called EFIGE (pronounced "effigy"), for specifying the complex relations needed to describe a distributed problem solving system's organizational structure. We also describe an interpreter for EFIGE, that is able to instantiate a particular organization by combining a description (in EFIGE) of a generic *class* of organizational structures and a set of instantiation constraints that specify the particular instantiation. The introduction of relations defined with constraints to organization descriptions significantly enhances the description as a symbolic representation of the organization. It allows the description of organizational *classes*, as opposed to descriptions of specific instances of some class. Constraints, however, complicate organization instantiation. To instantiate a relation, solutions must be found that satisfy each of the constraints in the relation. This requires searching large spaces of possible solutions in an attempt to find values that will simultaneously satisfy all of the constraints. As an interim approach, we have adapted an algorithm from the Artificial Intelligence literature that is used to eliminate inconsistent assignments of values to constraints [7]. This approach is limited, however, because it tries to choose solutions that satisfy one constraint without first performing some analysis that will insure that the solution will be acceptable to the remaining constraints. The use of a more sophisticated approach awaits further research.

In the next section we present an example of an organizational structure, then discuss how it might be described in an organizational description language. Section 3.3 introduces the description language EFIGE and indicates how structures are described within this language. Section 3.4 describes how descriptions are instantiated, and Section 3.5 discusses the current status of this work and its relation to ongoing research.

3.2 An Example

In this section, a *hierarchical* organizational structure for distributed signal interpretation is presented. We use this organization as an example with which to identify organizational features requiring description.

In our scenario for distributed signal interpretation, different kinds of signals are emitted by various vehicles as they move through a region. The system's task is to create a history of vehicular activity within the region based on the signals it detects. One processor organizational structure for performing signal interpretation is the *hierarchical* organization. It has three types of components: *sensing nodes*, which perform signal detection and classification; *synthesizing nodes*, which make local interpretations of the signal information they receive from the sensing nodes; and *integrating nodes*, which combine interpretations received from other nodes to create interpretations over larger portions of the sensed region. Figure 3.1 illustrates an instance of the hierarchical organizational structure that has one integrating node, four synthesizing nodes, and four sensing nodes. The figure also shows the lines of communication between the nodes, although the directionality of these communication links and the information transmitted is not the same between all pairs of nodes. Finally, the figure indicates the overlapping regions scanned by each sensor. Figure 3.2 shows another instance of the hierarchical organizational structure. It has five integrating nodes, sixteen synthesizing nodes, and sixteen sensing nodes.

Figures 3.1 and 3.2 show two *instances* of the same organizational *class*. The goal of this work is to develop a way of describing organizational classes, as opposed to describing specific organizations that are instantiations of some class. The key features of any organizational class are the different types of components (in the distributed signal interpretation application, sensing, synthesizing, and integrating nodes) and the relations between them (communication between sensing and synthesizing nodes, synthesizing and integrating nodes, and low-level and high-level integrating nodes[1]). Each type of component has its own particular set of responsibilities to carry out (signal detection, interpretation, integration) and a set of requirements for resources to be utilized in meeting its responsibilities (processing hardware, knowledge about signal interpretation, etc.). The

[1]This last relation is not instantiated in Figure 3.1 because there is only one integrating node.

ıres are described within
ntiated, and Section 3.5
›ing research.

.ted signal interpretation
o identify organizational

.‹inds of signals are emit-
stem's task is to create
signals it detects. One
:ation is the *hierarchical*
ı, which perform signal
al interpretations of the
ntegrating nodes, which
erpretations over larger
: of the hierarchical or-
.esizing nodes, and four
. between the nodes, al-
nformation transmitted
dicates the overlapping
ınce of the hierarchical
ynthesizing nodes, and

ıtional *class*. The goal
sses, as opposed to de-
ss. The key features of
ı the distributed signal
ıodes) and the relations
ıodes, synthesizing and
). Each type of compo-
nal detection, interpre-
utilized in meeting its
erpretation, etc.). The

e integrating node.

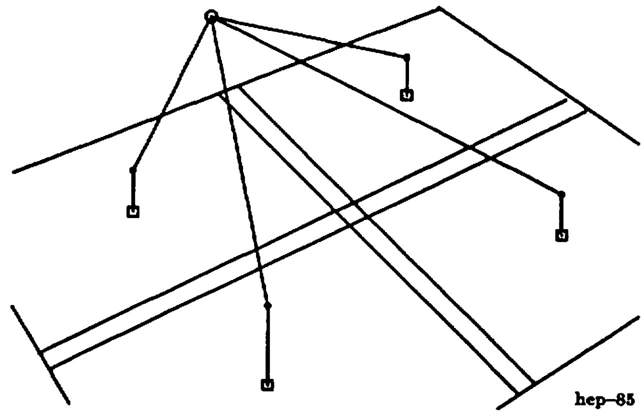

Figure 3.1: An instance of the hierarchical organizational structure with one integrating node (circle), four synthesizing node (dots), and four sensing nodes (squares).

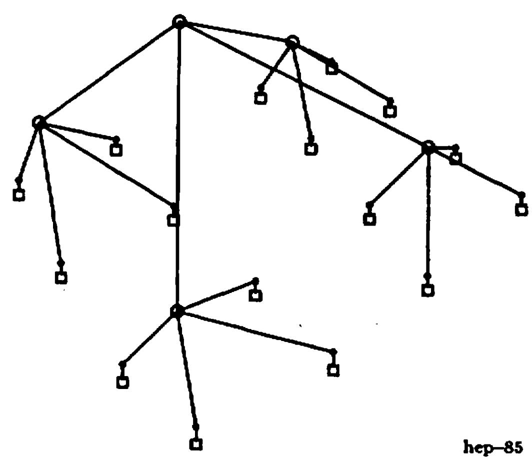

Figure 3.2: The hierarchical organizational structure with five integrating nodes, sixteen synthesizing nodes, and sixteen sensing nodes.

relations between component types are independent of the numbers of components that may be instantiated for each type or on what processor they may execute—synthesizing nodes must always receive signal information from sensing nodes. For that reason, their descriptions must also be independent of details specific to single instances of the organization.

The key features of an organizational class are its components and the relations between them. In the rest of this section, we indicate what information must be included in a description of these organizational features and the range of values that will have to be accommodated. We start, however, with a discussion of the organization's *purpose*.

3.2.1 Purpose

An organization is a group of one or more individuals whose purpose is to perform some set of tasks in an attempt to achieve a set of goals while observing a set of constraints. Constraints on how the goals are to be achieved determine the rate of processing needed and, in turn, affect the size and complexity of the organization. For example, the goal of the hierarchical organization is to create a high-level history of vehicular activity over a region. The tasks required to achieve the goal include the detection and classification of acoustic signals generated by the vehicles, the weighing of evidence for the presence of a particular type of vehicle based on the signal types detected, and determining the paths of vehicles through the region and recording them.

Constraints on achieving the organization's goal emphasize processing tradeoffs between such features as topicality, production costs, robustness, completeness, and quality. For example, in the signal interpretation task, we may insist that the system produce highly rated interpretations of the data as quickly as possible, thus emphasizing maximal values for topicality (short response time) and quality (correct interpretations), at the expense, perhaps, of production costs (the rate of processing needed to derive the answer). Furthermore, distributed systems are typically expected to be robust; able to adjust to node failures and to have performance degrade gracefully as error in the system increases.

3.2.2 Components

Organizations are composed of components. The hierarchical organization, for instance, has three components: sensing, synthesizing, and integrating nodes. What these components have in common are sets of *responsibilities* and *resources* to be used in meeting them.

Responsibilities

Components perform tasks. These include: a subset of the tasks necessary for accomplishing the organization's purpose; management tasks incurred as organizational overhead; and—especially in human systems—tasks that counter, or do not contribute towards, the organization's purpose but are, for idiosyncratic reasons, important to the component. One way of specifying responsibilities is by assigning components subregions of the problem-solving space defined by the organizational task. For the signal interpretation task, the dimensions of the problem-solving space might be the physical region monitored by the system, problem-solving events (such as the detection of a signal of a certain type, the decision that a group of signals were produced by a particular type of vehicle, etc.), abstraction levels (signals of different types, groups of signals, vehicle types, patterns of vehicles), and time. Out of all of the tasks that an organization for signal interpretation needs to perform to meet its goals, sensing nodes perform only the signal detection task. Other components are responsible for performing the remaining tasks.

Resources

Components possess certain resources with which they are expected to perform their tasks, thus the resources required by a component will depend on the roles it plays in the organization. We describe three "flavors" of resources: software resources (knowledge), hardware resources (tools), and other components (consultants). Access to a component resource is access to another set of software and hardware resources and another list of component contacts.

Knowledge. We also describe three types of knowledge: **algorithms**, **data bases**, and **expertise**. Algorithms specify how to process data, data bases are repositories of information, and expertise refers to the type of heuristic knowledge characteristic of expert systems. The problem-solvers located at each node may incorporate any or all of these forms of knowledge. Algorithms and expertise, for example, tell a node how to interpret signal data as evidence for the presence of vehicles and how to track those vehicles. Some knowledge may be meta-level knowledge used to determine when it is appropriate to apply domain-specific knowledge.

Tools. In addition to knowledge about how to perform a task, a worker may require particular implements with which to execute the task. These can be effectors (a robot arm or the hammer or wrench that the arm may wield during a particular process) or sensors (the devices that a sensing node uses to detect signals). Use of a tool requires that the worker have additional knowledge: how to use it.

Consultants and Subcontractors. If unexpected problems arise that are outside the range of expertise of a component, it is useful to know of someone who does have the expertise. Given this information, the component could ask for problem solving advice or contract the problem's solution to the expert. Similarly, a component might find it useful to know who can use its data, who can provide it with missing data, or who is available to share its processing load. Smith has investigated a method of distributed problem solving, called the contract-net approach, in which a node, given a problem that it cannot solve alone, contracts for the solution of the problem or of its subproblems [8]. This method does not rely on knowing in advance who is capable of solving the problems or subproblems, since they can be broadcast to the network, but this information is used if available. This is known as *focused addressing*. We can imagine a scenario in the signal interpretation task in which a sensing node begins sending a synthesizing node information about signals of a type for which the node has no knowledge. If the synthesizing node knows, however, of another node that does have the knowledge, it could ask for help. If not, it could broadcast a request for the knowledge it needs.

Individual Characteristics

There may be information about a component that is not directly related to its responsibilities or resource requirements. For instance, it may be necessary to have some abstract description of how the component will function, especially if the organization's performance is to be evaluated before instantiation. The level of detail will vary with the application, but can include estimates of the average reliability of the component's outputs, mean time to failure, rates at which inputs can be processed, or even a state transition model that simulates how the component will behave. Pavlin, for example, presents a way of modeling the behavior of an entire distributed problem solving organization [9].

3.2.3 Relations Between Components

Components in an organization do not exist, nor do they function, independently of one another. Components interact. Commands, information, and subassemblies (including partial solutions) are passed between them, and they may work cooperatively at performing operations on some object. These interactions are expressed as relations between the components involved.

Relations between components can be arbitrarily complex. It will seldom be the case that only a single relation will exist between only two components. In general, a conjunction of relations between groups of components will be required. These groups may, in turn, be formed from other relations.

Communication

The most important relation between two or more components is who talks to whom. This is the relation shown most prominently in Figures 3.1 and 3.2, where each internode line represents an instance of a communication relation. The communication relation is used to identify a component's sources of a particularly valuable resource, ***information***, and to identify the consumers of the information it produces.

Equally important are the details of what is exchanged during communication. The need to associate a message structure with a communication relation complicates its instantiation. It requires that objects satisfying the relation must, additionally, satisfy the constraint that their message structures be compatible. That is, if one object expects to send messages consisting of certain information in a specific format, the other object in the relation (assuming the binary case) must be prepared to receive that information in the same format.

Finally, it may be necessary to associate a specific communication strategy with a communication relation. Durfee, Lesser, and Corkill have investigated the effects of several communication strategies on the global behavior of a distributed problem solving network [10].

Authority

Authority is a relation that indicates how much emphasis should be given to messages from different sources or, possibly, to different messages from the same source. If the message has authority, the component may allow it to have greater impact on its activities. In the five-node organization, the integrating node may be given the authority to direct synthesizing nodes to look for evidence of vehicles in regions it designates. Upon reception of such a message, a synthesizing node might cease whatever processing it had chosen to do (based on the local information available to it) and take up the requested work.

How much attention should be paid to an authority? The component may realize that the environment has changed and the authority's instructions are no longer appropriate. Should they be followed, ignored, or disputed? A synthesizing node may have very strong evidence that a vehicle's path lies in a certain direction when it receives a directive from the integrating node to look elsewhere. The node must decide if it is more important to continue processing the strong data or to follow the integrator's instructions. In fact, it may be desirable to have individual variation between nodes, weighting some synthesizing nodes with greater bias toward the integrating node's authority than others. Nodes with little bias towards authority are called self-directing or *skeptical*. Reed and Lesser have discussed the importance of self-direction in the members of honey bee colonies [11]. Corkill discusses the use of skeptical nodes in distributed problem solving organizations performing signal interpretation [12].

In general, organizational relations can be described on two levels, at a (relatively) global level outlining the relation and its participants, and at the local level, where details and individual variance are elaborated.

Location, Proximity, et cetera

Many other important relations may exist between the components of an organization. For instance, if one component is a producer of a product whose value decreases with time, the component using that product may need to be located nearby, or they may both need to be placed near terminals of a reliable transportation network. Sales offices for a manufacturer may need to be located across the country, instead of all in one city. Sensing nodes in the organizations for signal interpretation need to be distributed across the entire region. Synthesizing nodes need to communicate with a sensing node (more generally, group of sensing nodes) that scans the nodes' region of responsibility.

3.2.4 Composite Components

Organizations are often composed of suborganizations. In order to simplify descriptions of such organizations, the suborganizations are treated as single components and the interactions among these components are detailed; then the components are "enlarged" to reveal the suborganization they represent. While these *composite* components do not have physical counterparts in the actual organization, they serve two purposes: they help make descriptions of organizations understandable, and they group physical components that perform the same organizational function. For these reasons, an organizational description language should provide the ability to logically "package" an organization as a single component of another organization. Furthermore, the language should treat individual and composite components the same. If one description knows as little as necessary about another, it will be easier to make modifications.

Composite components allow recursive descriptions of organizations. If there are enough nodes (twenty-one, for instance), the hierarchical organization (Figure 3.2) is instantiated as an integrating node with hierarchical organizations as its components. Each of these hierarchical suborganizations is again instantiated with its share of the original nodes. When the number of nodes becomes small enough, the organization is instantiated as a single integrating node with synthesizing nodes under it. If the number of nodes is small enough to start with, of course, no composite components need to be created. The synthesizing nodes are created right away. This is the case for five nodes, for example.

3.3 Describing Organizational Structures with EFIGE

This section introduces EFIGE, a language for describing organizational structures. Descriptions of organizations in EFIGE are hierarchical. That is, they are composed of either individual or composite structures, and a composite structure's components may be individual or composite. Figures 3.3 and 3.4 show part of the description of the hierarchical organization presented in Figures 3.1 and 3.2. Descriptions have global names, parameters that may have default values, and local variables. Components are given local names, are conditionally instantiated, may be replicated, and information—in the form of values for parameters—may be partitioned among them. Parameterized descriptions and conditional instantiation of components allow descriptions to be defined recursively. This is the case with the hierarchical organization.

Figure 3.5 shows part of the description of an individual component. Fields are provided for specifying the individual's duties within the organization, listing the resources the individual will require to meet its duties, and for additional information about the individual that may be accumulated during instantiation or may provide information to be used to estimate the individual's processing characteristics. Values for these fields are necessarily application dependent.

3.3.1 Relations

The hierarchical approach we have presented for describing organizations is similar to the specification framework of other languages. What gives our approach additional representative power is the introduction of relations and constraints into this hierarchical descriptive framework.

EFIGE allows relations of any kind to be established between components and allows additional information to be associated with the relation. For instance, almost all languages for describing organizational structures give their individual and composite struc-

```
;; All descriptions are given names.

(NAME hierarchical

 ;; A 'composite' description has components.

 TYPE composite

 ;; Descriptions are parameterized.  The user can specify that a
 ;; parameter be bound to a different value than its default.

 PARAMETERS
   ((number-of-integrating-nodes  :DEFAULT 5)
    (region                          ...    ))

 ;; The LOCAL-VALUES field is used to compute and assign values to
 ;; local variables.

 LOCAL-VALUES
   ((number-of-synthesizers  ...  )
    (number-of-hierarchies   ...  ))

 ;; The COMPONENTS field lists the components of a composite
 ;; organization.

 COMPONENTS

   ;; Components are given local names.

   ((COMPONENT-NAME synthesizers

     ;; Components are described by other organizational descriptions.
     ;; A description called 'synthesizing-node' is used to describe this
     ;; component.  It could be used to describe other components, as well.
     ;; 'Synthesizing-node' has a parameter, 'region', which will be set to
     ;; the value of 'worker-region' (defined in the COPIES field).
```

Figure 3.3: Part of the Description of the Hierarchical Organization

```
DESCRIPTION (synthesizing-node ((region worker-region)))

;; The organization that describes a component may be instantiated
;; more than once, depending on the value in the COPIES field.
;; 'Synthesizing-node' is to be instantiated 'one-less-node' times.
;; 'One-less-node' is a local variable defined in the LOCAL-VALUES
;; field.

COPIES (one-less-node

        ;; The VARY clause of the COPIES field is a construct for
        ;; declaring variables and assigning them a sequence of values
        ;; 'Worker-region' will be assigned a different value for each
        ;; instantiation of 'synthesizing-node'; consequently, each
        ;; instantiation will have a different value for its 'region'
        ;; parameter.

        (VARY
          (worker-region  ... )))

;; Components are only instantiated if their PRECONDITION predicate
;; function evaluates to true.  This component is to be instantiated
;; only if 'number-of-nodes' is within the range 3-5, inclusive.

PRECONDITION (within-subrange? number-of-nodes 3 5))

;; A second component.

(COMPONENT-NAME subhierarchy

 ;; The component, 'subhierarchy', is described by the description,
 ;; 'hierarchical', thus this component is recursive.

 DESCRIPTION  (hierarchical  ... )
 COPIES       (number-of-hierarchies
                (VARY  ... ))
 PRECONDITION (> number-of-nodes 5))
... )
... )
```

Figure 3.4: More of the Description of the Hierarchical Organization

```
;;; This description is of an individual structure, it has no components :::

(NAME synthesizing-node
 TYPE individual

 ;; Descriptions of individuals have PARAMETERS and LOCAL-VALUES fields,
 ;; but we'll ignore them here.

       ...

 ;; The tasks that an individual are to perform are specified in the
 ;; RESPONSIBILITIES field.  For our application, responsibilities are
 ;; specified as regions of the problem-solving space and rated by
 ;; importance.  'S1-sensor-regions' is bound to a description of a
 ;; problem-solving region in the LOCAL-VALUES field.

 RESPONSIBILITIES
   ((PROCESS-AREA (s1-sensor-regions)
     IMPORTANCE       ...   )
        ...      )

 ;; The resources the individual needs to perform the tasks for which it is
 ;; responsible are given in the RESOURCES field.  One resource required by
 ;; our application is knowledge about specific tasks.

 RESOURCES
   (KNOWLEDGE-SOURCES
     ((KS-NAMES (determine-communication-kss ?this-description)
       GOODNESS      ...      )
        ...   ))

 ;; The CHARACTERISTICS field contains information that will vary
 ;; between individuals--even though they belong to the same component of
 ;; the organization--or information that can be used to simulate the
 ;; individual's behavior.

 CHARACTERISTICS
   (LOCATION   ...  )
    ...  )
```

Figure 3.5: Part of the Description of an Individual

tures *ports* and allow the composite structures to specify *communication links* among the ports of their components and between ports belonging to the composite structure and its component ports. But a communication link is only one kind of relation and ports are just devices for associating message structures, directionality, and other information with the relation. These concepts have been generalized in EFIGE.

There are three parts to the description of a relation and each part appears within the description of a different structure. The *declaration* of a relation between components appears in a composite structure (Figure 3.6). A declaration simply specifies that a relation exists between one, or more, components. Either type of component (individual or composite) can participate in a relation, but it is more likely that a composite structure will *forward* membership in the relation to some of its own components instead (Figure 3.7). Forwarding may occur again if the component to which membership in a relation is forwarded is another composite structure. Finally, the relation is *refined* within the structures that are to actually participate in it (Figure 3.8). This is where the constraints are specified and it is here, also, that any additional information is associated with it.

It should be noted that one relation may depend on the instantiation of another. For example, an integrating node may wish to communicate only with synthesizing nodes that receive information from sensing nodes with particular characteristics. This requires that the `sensor-synthesizer` communication relation be instantiated before the `integrator-synthesizer` relation. Because EFIGE is currently unable to recognize such situations, relations must indicate the order in which they are to be evaluated, relative to all of the other relations in the organization. This also helps the user avoid making circular references in constraints. The evaluation order of a relation is specified with its declaration.

A `RELATIONS` field is part of both composite and individual structure descriptions. It contains a list of parts of relations, although only refinement parts can appear in descriptions of individuals. Figure 3.6 shows the declaration part of a relation in EFIGE. The `RELATION-NAME` field gives the relation a local name; the `RELATION-TYPE` field indicates the type of relation expression. (The value `new` is used to indicate the declaration part of a relation, `forward` indicates the forwarding part, and `refine` the refinement part.) These two fields appear in all parts of the description of a relation. The integer expression in the `EVALUATION-ORDER` field is used to establish a partial order among `new` relations. The relations will be sorted in increasing order by their values for this field.

The `RELATE` field declares a relation between components by listing them as members of the *domains* of the relation. An n-ary relation has n domains. Each domain is provided with a name; component names, paired with the name of one of their relation

```
RELATIONS

  ;; Entries in the RELATIONS field are given names.

  ((RELATION-NAME sensor-synthesizer

    ;; Entries of type 'new' are used to declare the existence of a
    ;; relation between components of an organization.

    RELATION-TYPE new

    ;; This new relation is to be among the first implemented.

    EVALUATION-ORDER 1

    ;; This relation has two domains.  The first is given the name,
    ;; 'sensor', and consists of the structures instantiated for the
    ;; component, 'sensor-array'.  Within those structures, more
    ;; information about the relation is contained in an entry in
    ;; their RELATIONS field with the name, 'to-synthesizer'.
    ;; Similarly, the second domain is named, 'synth', and its
    ;; members are the structures instantiated for the 'synthesizer'
    ;; component.  These structures contain an entry in their
    ;; RELATIONS field with the name, 'to-sensor', that also
    ;; contains more information about the relation.

    RELATE ((sensor sensor-array$to-synthesizer)
            (synth  synthesizers$to-sensor)))
      ...
  )
```

Figure 3.6: Example of the Declaration of a Relation

```
(RELATION-NAME middle-integrator

 ;; Composite structures may have entries in their RELATIONS field
 ;; with type, 'forward'.  These pass the composite structure's
 ;; membership in a relation on to one (or more) of the structure's
 ;; components.

 RELATION-TYPE     forward

 ;; The structures instantiated for the 'integrator' component will
 ;; become members in the relation in place of the composite structure.
 ;; The entry in their RELATIONS field with the name, 'upper-exchange',
 ;; will contain more information about the relation.

 FORWARD           (integrators$upper-exchange))
```

Figure 3.7: Forwarding a Relation

parts, are listed after it. All copies of the component will be included in the domain. The relation parts in the components must either refine the relation or forward it. The relation `sensor-synthesizer` in Figure 3.6 has two domains named `sensor` and `synth`. The members of the `sensor` domain are all of the copies of the structure instantiated for the component `sensor-array`. Similarly, the members of the `synth` domain are the structures instantiated for the `synthesizers` component. Within these structures, there must be an entry in their respective `RELATIONS` fields named `to-synthesizer` and `to-sensor`, respectively.

Figure 3.7 shows an example of the forwarding of a relation. In effect, forwarding a relation results in the replacement of the reference to a composite structure in the original relation with the list of the composite structure's components. Thus the structures instantiated for `integrators` will receive membership in the relation instead of the structure which includes `middle-integrator`.

A `refine` expression is embedded in the structure that will participate in the relation. It contains constraints for refining the relation and additional data that is to be associated with the relation. Constraints are discussed below. Figure 3.8 gives an example of relation refinement. The relation part `to-sensor` appears in the individual structure `synthesizing-node` which was instantiated as the `synthesizers` component of the composite structure `hierarchical` (Figures 3.3 and 3.4). It refines the relation `sensor-synthesizer`, which referred to it in Figure 3.6. Since this is (implicitly) a communication relation, `to-sensor` includes information that is to be associated with the relation (such as the direction messages are to travel in the relation, their format, communication strategies, etc.).

3.3.2 Constraints

EFIGE allows each member of a relation to make local refinements to the relation's domains using a combination of restriction, group, and preference constraints. A relation, R, defines a set of n-tuples, $(x_1 \ldots x_n)$, that is the Cartesian product of n (not necessarily distinct) sets, $X_1 \times \ldots \times X_n$ (the domains of the relation). The number of n-tuples is equal to the product of the cardinality of each set. EFIGE uses restriction, group, and preference constraints to reduce the size of each of these sets and, hence, the size of R. They are described in this section.

Restriction

Restriction constraints are applied to the members of a set to identify those members for which the constraint evaluates to `true`. In other words, the constraint acts as a characteristic function, identifying a new set among the members of the old. EFIGE allows such a function to be provided for all of the domains of a relation:

$$(P_i\, X_i), i = 1 \ldots n$$

where

(FX) denotes the set $\{x | (x \in X) \wedge (Fx)\}$,

X_i is the i-th domain of R,

P_i is a predicate over X_i: the constraint. In effect, the relation becomes:

$$R = \prod_{i=1}^{n} (P_i\, X_i)$$

where $\prod_{i=1}^{n} X_i$ is used to indicate the Cartesian product, $X_1 \times \ldots \times X_n$.

Restriction constraints are used to identify those members in the other domains of a relation that are acceptable to the current member of the current domain as partners in the relation. The TASK language uses restriction constraints to direct assignment of resources [6]. These are limited to specification of proximity relations between processes and sets of physical resources identified by their attributes (features of the Cm* hardware). A TASK constraint, for example, might specify that a process must execute on a processor

```
(RELATION-NAME to-sensor

 ;; Each of the ultimate members of a relation (after all forwarding of
 ;; membership) has an entry of type 'refine' for that relation.
 ;; The 'refine' entry may provide each member with fields for the
 ;; description of additional information needed by the relation and may
 ;; reduce the size of the relation by allowing each member to reject
 ;; some of the tuples in which it was included when the relation was
 ;; originally defined (with a 'new' relation entry in the description
 ;; of some composite structure).

 RELATION-TYPE refine

 ;; The CONSTRAINTS field contains the constraints with which tuples
 ;; in the relation are selected and/or rejected (see Figure 9).

 CONSTRAINT
    ...

 ;; The ADDITIONAL-DATA field is used to add information to a
 ;; structure's description that is needed for the relation.  A
 ;; communication relation, for example, needs to know the direction
 ;; in which messages will travel, the type of message, and a
 ;; description of its format.

 ADDITIONAL-DATA
   ((communication
      ((DIRECTION  receive
        NATURE     (hyp)
        DISPATCHES ... ))
          ...  ) ... )
     ...   )
```

Figure 3.8: Relation Refinement

with a large local memory. Artificial Intelligence programs that perform planning tasks also use restriction constraints. For example MOLGEN, when planning experiments in molecular genetics, generates a constraint restricting the choice of a bacterium to one that resists an antibiotic [13].

Group

Group constraints are applied to a single set to create a set of sets. Each set in the new set is a subset of the original and, for each, the constraint evaluates to **true**. Thus the constraint is a characteristic function with a domain that is the power-set of the original set. As with restriction constraints, EFIGE allows a group constraint to be specified for each domain of a relation:

$$(Q_i\, \mathcal{P}(X_i)), i = 1 \ldots n$$

where

$\mathcal{P}(X)$ denotes the power-set of X,

Q_i is a predicate over $\mathcal{P}(X_i)$.

The group constraint, Q_i, identifies a set of sets: each subset, or group, is acceptable as the i-th domain of the relation. Thus alternate relations are possible, one for each combination of groups from each domain:

$$R = \prod [\bigvee [\prod_{i=1}^{n} (Q_i\, \mathcal{P}(X_i))]] \tag{3.1}$$

where $\bigvee X$ is used to indicate that alternative selections can be made from X and $\prod X$ denotes the Cartesian product of an indeterminate number of sets, the members of X.

Group constraints identify groups of objects that together satisfy some property that their individual members cannot (unless the size of a group is one). For instance, a relation in an organization that performs distributed signal interpretation may specify that sensing nodes are to communicate with synthesizing nodes. Each synthesizing node may use a group constraint to refine the relation by requiring that it communicate only with groups of sensing nodes that together can provide information about the entire region for which it is responsible. ADABTPL, a language for describing databases, employs both group constraints and restriction constraints [14].

Preference

Preference constraints implicitly define a partial order over a set by selecting one object from it. If this object is then removed, a second may be chosen, and so on. The i-th object in the ordering over a set X, where S is the preference constraint, is $(S\ V_i)$, for $1 \leq i \leq |X|$, where

$$V_i = V_{i-1} - \{(S\ V_{i-1})\}$$

$$V_1 = X$$

$$V_{|X|+1} = \phi.$$

Preference constraints may be used by any member of a relation to refine any domain:

$$(S_i^k\ X_i), i = 1 \ldots n, k \in \{1 \ldots |X_i|\}$$

where $(S^k X) \equiv (SV_k)$. Using preference constraints alone reduces the relation to a single tuple:

$$R = ((S_1^{k_1}\ X_1) \ldots (S_n^{k_n}\ X_n)), k_i \in \{1 \ldots |X_i|\}$$

During instantiation, preference constraints are employed to choose between the apparently equal options generated by a group constraint. For instance, the group constraint refining the communication relation between sensing and synthesizing nodes may identify two groups of sensing nodes that will be acceptable to a synthesizing node. A preference constraint is used to choose between them. The smaller group may be chosen in order to reduce communication overhead.

Composition

Composition provides a means for functional composition. Thus the domain of one constraint may be a set that has been defined by another constraint and the domains of a relation may be refined by many constraints. For instance:

$$R = \prod_{i=1}^{n} (S_i^{k_i}\ (Q_i\ \mathcal{P}(P_i\ X_i))), k_i \in \{1 \ldots |X_i|\} \tag{3.2}$$

where

S_i is a preference constraint,

Q_i is a group constraint,

P_i is a restriction constraint,

X_i is the i-th domain of R.

Note the differences between Equations 3.1 and 3.2. Preference constraints identify a single group from the list of groups produced by each group constraint, with the result that a single relation is selected from among the myriad possibilities.

EFIGE allows each member of a relation to refine (further specify) the relation using restriction, group, and preference constraints that are composed with each other in that order. The resulting constraint is evaluated for each member in its local context; thus the results may vary from member to member, even though the same constraint is applied. In any case, the solution to the overall relation then becomes (approximately) the union of the results of applying each of its members' local refinements:

$$R = \bigcup_{j=1}^{m}[\prod_{i=1}^{n}(S_{ij}^{k_j}\ (Q_{ij}\ \mathcal{P}(P_{ij}\ X_{ij})))]$$

where

$m\ = \sum_{i=1}^{n}|X_i|$ (i.e. j varies over all of the members of the relation),

$k_j\ \in \{1 \ldots |(Q_{ij}\ \mathcal{P}(P_{ij}\ X_{ij}))|\}$,

$X_{ij}\ = X_i - \{j\}$,

F_{ij} is member j's constraint on domain i.

Actually, the union operator is too simple for combining the local refinements to a relation. This is because the tuples in the desired relation must be *consistent* with one another. If, for example, the constraints for a member j of a binary relation select a tuple $(j l)$, then the constraints for l should include that tuple in their selection as well. If this is not the case, it may be that the two members can be made consistent by using their second choices for tuples (choosing different values for k_j). Section 3.4 presents an algorithm for evaluating constraints and combining them in such a way that they are consistent.

Figures 3.9 and 3.10 show an example of the constraints a synthesizing node might use to refine a communication relation with sensing nodes. The `PARTNERS` field lists the names of the domains in the relation, omitting the name of the domain to which the synthesizing node belongs. The names in this list must match those given when the relation was declared (except for the name of the domain in which this constraint is a member). A restriction, group, and preference constraint must be provided for each of the domains listed. Thus the `RESTRICTIONS`, `GROUPS`, and `PREFERENCES` fields each contain a list of ordered pairs: the name of the domain followed by the constraint that will be applied to it. The restriction constraint is applied first to all of the members of a domain.

In Figures 3.9 and 3.10, the remaining contraints insure that the information associated with the relation (in the ADDITIONAL-DATA field, see Figure 3.8) is compatible and that the sensing node detects signals in at least part of the region for which the synthesizing node is responsible. Since this is a communication relation, compatibility means that there must be at least one sender and at least one receiver in the relation and that the proposed topics for discussion overlap. The group constraint is applied to those members that satisfied the restriction constraint. In this example, it will form groups of sensing nodes that together detect signals over the specified region. The preference constraint is applied to the groups to select one of them. In this case, it will choose the smallest group.

3.3.3 The Procedural/Declarative Interface

Figures 3.9 and 3.10 illustrate that much of the information in a description written in EFIGE is procedural. That is, functions provide details about how an organization is to be instantiated. This information is inherently application dependent; users of the language will need to develop libraries of the functions useful for each application. For instance, in Figure 3.9, the function, `sensors-that-cover-region`, returns groups of sensing nodes that, together, are able to detect signals from every part of a rectangular region. This function will not be of use in most applications. The declarative part of EFIGE, the component fields, provide a framework for organizing the procedural information and a method for applying it. Appendix B describes other functions needed to describe organizations for our distributed signal interpretation application.

3.4 Instantiating a Description

Instantiating an organization involves performing parameter substitution, testing component preconditions to find out which are to be instantiated, instantiating each component the specified number of times with the indicated parameter settings, and implementing relations between components. Implementing a relation requires finding solutions to each of the constraints associated with the relation. Finding these solutions is difficult because the solutions may interact. For example, the constraints refining a communication relation between synthesizing nodes and sensing nodes may choose the same sensing node for each of three synthesizing nodes. The sensing node's constraint's, however, may restrict it to communicating with any two of the synthesizing nodes, but not all three, in order to limit the amount of time it must allocate to communication. One of the synthesizing nodes will have to choose a different sensing node, which may affect the choices of other

```
CONSTRAINT

  ;; The PARTNERS field lists the other domains of the relation (other
  ;; than the one to which the owner of this constraint belongs).  A
  ;; constraint of each type is provided for each of the domains.

  (PARTNERS (sensor)

   ;; The RESTRICTIONS field predicates act as filters, rejecting
   ;; members of the other domains that do not meet their criteria.

   RESTRICTIONS

     ;; The predicate 'compatable-communication?' examines the
     ;; descriptions in the ADDITIONAL-DATA fields of this relation
     ;; (see Figure 8) and each member of the 'sensor' domain for
     ;; consistency (e.g., since the DIRECTIONS field in this relation
     ;; is 'receive', the other must be 'send').

     ((sensor (and (compatable-communication?
                     ?this-relation ?partner-relation)

     ;; This predicate determines if the area scanned by each sensing
     ;; node includes the area specified by region.  The symbol
     ;; '?partner-structure' will be bound to each sensing node's
     ;; structure description.  In contrast, the symbol
     ;; '?partner-relation', above, is bound to the relation entry in
     ;; each structure description that is used to refine the relation
     ;; between sensing and synthesizing nodes.

                   (sensor-scans-part-of-region?
                    region ?partner-structure))))

   ;; The functions in the GROUPS field select groups of tuples in
   ;; which the members of the given domain are, together, able to
   ;; satisfy some predicate.  The function,
   ;; 'sensors-that-cover-region', returns a list of those groups of
   ;; non-redundant sensors that together are able to scan the area
   ;; given by 'region'.  The symbol, '?candidate-structures', is
   ;; bound to a list of the the descriptions of those structures
   ;; that passed the restriction constraints.
```

Figure 3.9: Constraints

```
GROUPS ((sensor (sensors-that-cover-region
                      region ?candidate-structures)))

;; The functions in the PREFERENCE field return one of the
;; groups of tuples formed by the group constraints.  The
;; function, 'select-smallest-set' finds the group with the
;; least number of members.

PREFERENCE ((sensor (select-smallest-set ?groups))))
```

Figure 3.10: Constraints (continued)

nodes. In this section, we first present the algorithm for finding solutions to constraints, then briefly describe how the `hierarchical` organization is instantiated.

3.4.1 The Constraint Solution Algorithm

The algorithm we use for finding solutions to the interacting constraints associated with a relation first applies each member's preference and group constraints, then chooses a member with the smallest number of groups. Thus a synthesizing node whose group constraint produced only one solution will be processed before any node with two or more groups to chose from. This strategy minimizes branching in the search tree, which is important because we have no global knowledge to apply when choosing a branch. Instead we use local knowledge. The member's preference constraint is used to select one of its groups, if there is more than one. The selected group is a *local* solution. Local solutions are then used to build the global solution. The local solution lists the sensing nodes with which this synthesizing node will communicate, the global solution contains all of the sensing-synthesizing node pairings.

The groups of the other members of the relation that do not yet have a local solution must be made *consistent* with the solution just chosen. For the other members' groups to be consistent with the solution they must either:

1. contain the name of the member just processed, if the solution contains their name;
2. not contain the name of the member just processed, if the solution does not contain their name.

Inconsistent groups are deleted and the unprocessed member that now has the smallest group is selected for processing. Thus the choice of a local solution may prune the search tree and affect the order in which nodes in the tree are visited.

If any of the other members has all of its groups deleted, a new group must be chosen for the local solution, the effects of making the other members consistent with the old solution undone, and they must be made consistent with the new solution instead. If all of a member's groups are tried as local solutions without success, chronological backtracking is employed. The search is returned to the last member processed, its local solution is discarded, its consistency effects undone, and so on. If the search ends up back at the first member tried and tries all of its groups unsuccessfully, no global solution exists and the relation cannot be implemented.

The complete algorithm is included in Appendix A.

3.4.2 Instantiation of the Hierarchical Organization

Figure 3.11 shows how instantiation of each composite description leads to instantiation of individual components and the implementation of relations between them. The `hierarchical` organization was instantiated with the `number-of-nodes` parameter set to twenty-one and the `number-of-sensors` parameter set to sixteen. When the upper `hierarchical` structure was instantiated, the preconditions of only two of its components evaluated to true: the `integrators` component and the `subhierarchies` component. One copy of the `integrators`, and four of the `subhierarchies`, were instantiated. The `integrator-integrator` relation is implemented because, at this point, it is actually an `integrator-subhierarchies` relation. In the `subhierarchies`, membership in the relation is forwarded to their `integrators` components.

Each of the `subhierarchies` components is another `hierarchical` organization. This time, however, in each of them the precondition for the `subhierarchies` component evaluates to false and the recursion stops. The other components' preconditions evaluate to true and, for each of the new `hierarchical` organizations, one `integrators`, four `synthesizers`, and four `sensor-array` components are instantiated. In each organization, an `integrator-synthesizer` relation and a `synthesizer-sensor` relation is implemented.

3.5 Status and Ongoing Research

This section describes the current status of EFIGE followed by a discussion of improving the organization instantiation process and a discussion on using organization descriptions to automate the configuration process. These activities are steps toward an eventual goal of *organizational self-design*, where the organization is able to reconfigure modify itself in response to changes in its operating requirements and environment.

group must be chosen
onsistent with the old
lution instead. If all of
nological backtracking
d, its local solution is
h ends up back at the
bal solution exists and

ation

ion leads to instantia-
is between them. The
-nodes parameter set
een. When the upper
two of its components
rarchies component.
were instantiated. The
is point, it is actually
s, membership in the

cal organization. This
erarchies component
ts' preconditions eval-
ns, one integrators,
ntiated. In each orga-
er-sensor relation is

iscussion of improving
ganization descriptions
ward an eventual goal
configure modify itself
ent.

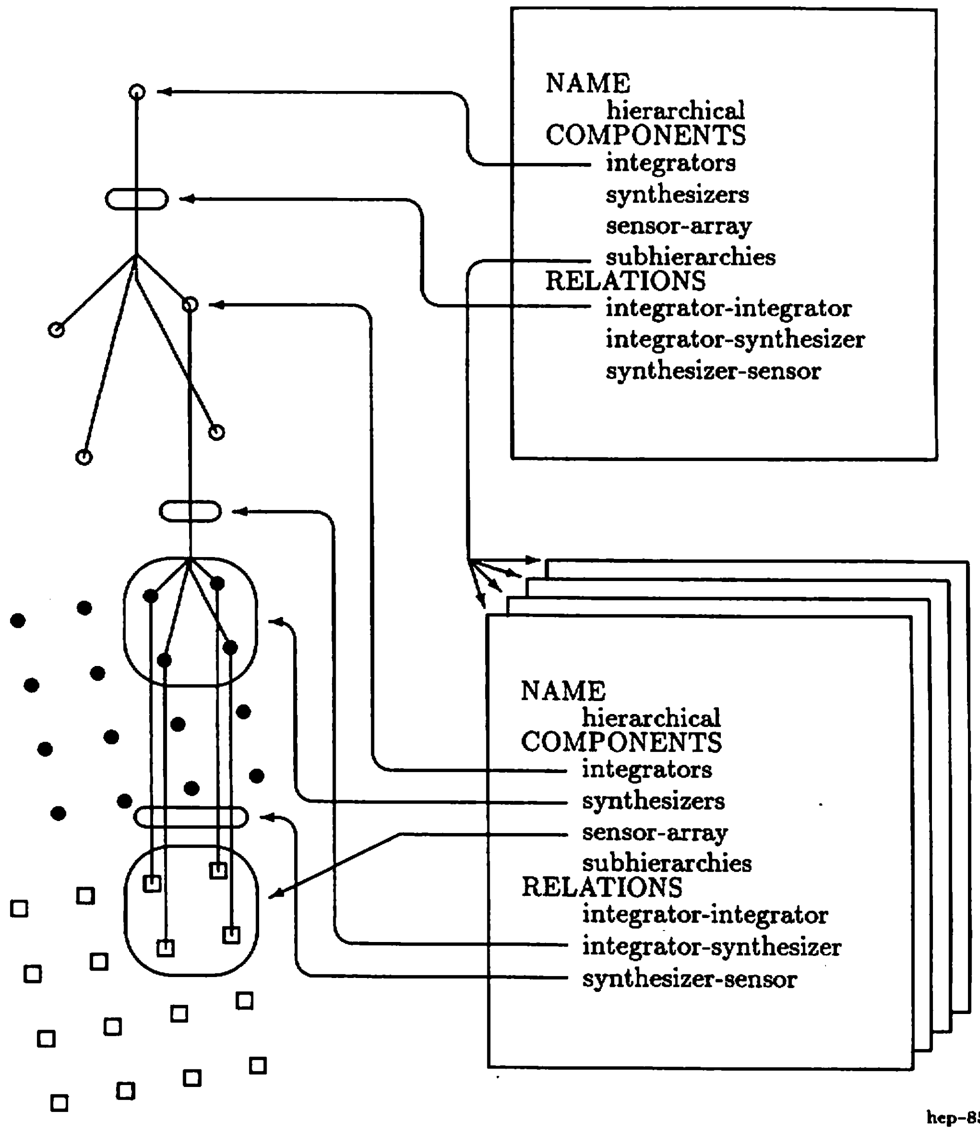

Figure 3.11: Instantiation of the hierarchical organizational structure with sixteen sensors, sixteen synthesizing nodes, and five integrating nodes, requires five instantiations of the `hierarchical` composite description as well.

3.5.1 Status

EFIGE has been implemented in Common Lisp. Descriptions have been written of organizational structures for use in the Distributed Vehicle Monitoring Testbed (DVMT) [15]. (The hierarchical organization used as an example in this chapter is one of these.) The DVMT simulates the execution of a distributed problem solver that performs signal interpretation. Descriptions of organizations in EFIGE are interpreted and added to a file of parameters that specify the experiment that is to be carried out on the DVMT. One description of an organization can be used to generate many instantiations of the organization by varying the values supplied to the description's parameters. This results in a savings in file space, since one description can be stored instead of many instantiations, and in the experimenter's time, because previous to this work instantiations had to be generated by hand—a time-consuming and error-prone procedure.

3.5.2 Improvements to EFIGE

Investigations directed toward finding answers to three questions should result in an improved system. These questions are:

1. How can the constraint mechanism be made more general?
2. How can search efficiency be improved?
3. What can be done when a set of constraints is over-constrained?

Directions in which to search for answers to these questions are considered in the following sections.

Bottom-Level Constraints

Currently, the EFIGE interpreter is free to physically locate nodes wherever dictated by the description and to assume that communication channels exist wherever needed. In effect, the system is allowed to configure the processing network as is convenient. This is useful for initially designing an organization outside the constraints of an existing architecture. The next step for enhancing the EFIGE interpreter is to allow it to instantiate the "best" organization of a specified class, given a particular network architecture. This involves specifying *bottom-level* constraints representing the particular network architecture and incorporating those fixed constraints into the instantiation of an organization. Bottom-level constraints specify that the instantiated organization include components with given values for some or all of their attributes or that particular relations be implemented. Such constraints could specify an entire processing network, making it the job of the interpreter to instantiate, as best as possible, an organization's functional components and their relations over a physical network that provides less than optimal support. For example, if bottom-level constraints specify that there are only a dozen processing nodes but the instantiated organization needs thirty-seven, the interpreter will have to assign multiple organizational nodes to the same processor.

Constraint Propagation

Because of the combinatorics, it may be unreasonable to apply restriction or group constraints to all of the members of a domain. For instance, the number of ways n synthesizing nodes can communicate with s sensing nodes, where any given synthesizing node may be assigned from zero to s of the sensing nodes, is 2^{ns}. The present algorithm attempts to avoid examining all of the objects of this set by eliminating subsets of objects on the basis of local information. Thus, if the restriction constraint for synthesizing node A selects sensing node P, P is checked to see if its restriction constraint selected A. If not, P is eliminated as a candidate for A. This eliminates from further consideration all of those configurations in which P and A are paired, thus cutting the search space in half. Unfortunately, evaluation of A's restriction constraint requires applying it to all of the sensing nodes in the domain and this is repeated for all of the synthesizing and sensing nodes in the relation.

Another approach to improving efficiency is constraint propagation [13]. In this method a description of the partner required by a member in a relation is gradually built up as constraints are evaluated. Constraint propagation, it is hoped, would allow the accretion of a more specific constraint that would identify, after only one pass say, the sensing nodes that both require and are required by a synthesizing node. Propagation of restriction constraints has been performed in systems such as MOLGEN [13]. Propagation of the more complex constraints used in EFIGE, however, is a problem that remains to be investigated.

Constraint Utility

When a set of constraints proves to be over-constrained, it would be useful to be able to intelligently modify them so that a solution can be obtained or to determine which ones must be satisfied and which ones can be safely ignored or relaxed. This requires knowledge about the purpose of the constraint (based on the organizational goals), so that judgments about its importance can be made, and it requires the ability to locate the conflict, to determine which constraints to modify. This may not always be possible. Fox assigns constraints *utility* ratings which can then be used to determine the usefulness of a given constraint's satisfaction, or lack of satisfaction, in a situation [16]. The least useful constraints are less likely to adversely affect results if they are not met. Utility ratings are also used during backtracking to find decision points where it is most likely that the wrong choice was made. A new choice is sought for and made at these points and the search is restarted.

Optimal Solutions

Group and restriction constraints provide binary valued ratings of choices: either an element of a set is accepted or it is rejected. Preference constraints order choices but provide no information about their relative worth. An assignment of relative worth to choices might allow more intelligent decisions to be made: several choices could turn out to be equivalent, or one choice may emerge as much more preferable than all others. The problem is, given the relative worth of local choices, how can they be optimized globally?

3.5.3 Organizational Self-Design

The long-term goal of this research is organizational self-design. An organization with this ability will perform the following tasks:

1. monitor the organizational structure's effectiveness in directing organizational activities,
2. identify new organizational structures appropriate to a new situation,
3. select the best among them,
4. implement the new structure over the network while preserving the network's problem solving activities.

This work's contribution toward organizational self-design is a language that provides for low-level, symbolic representations of organizational structures, but much work remains.

Organization Design

A slightly simpler problem is that of organization design. Organization design is the problem of choosing the best organization class—from a set of class descriptions—given knowledge about the organization's purpose (goal, task, and constraints on the goal) and the environment in which the organization is to operate. In fact, there are two problems: determining which organizations satisfy the constraints and then deciding which is "best". These correspond to steps 2 and 3 of the organizational self-design task.

Repairing Broken Organizations

Another simplification of the organizational self-design task is the problem of reconfiguration. Reconfiguration is needed to repair a "broken" instance of an organization (for example, one in which a component has failed), given its organization class description and environment information. This includes the problem of fault detection/diagnosis (roughly step 1), but the emphasis is then placed on recovering lost functionality without adopting a new organizational structure (eliminating steps 2 and 3, simplifying step 4). This is still a difficult problem; more fundamental problems underlie both it and the problems of organizational design and self-design. The sections below discuss some of these more fundamental problems. They also adopt a further simplification by considering static organizations (an organization capable of self-design is, by definition, dynamic).

Task Description

The purpose of an organization is to perform some task. A description of that task is essential for organizational self-design, and may be useful during instantiation as well. It is required for organization design in order to assign components their tasks, which will include parts of the organization's task. Fox states that tasks can be described by listing inputs, outputs, the transformations inputs undergo to become outputs, and the state transitions the processor goes through during task execution [3]. Is this information adequate for describing tasks? What is a suitable notation for representing this information? Pavlin, for instance, uses a Petri-net inspired approach to model the behavior of distributed problem solving organizations, this method might be adapted to describe tasks as well [9].

Organizational Goals

The goals of an organization are its desired performance abilities. Examples of organizational goals include: meet a minimum production rate, do not expend more than can be recovered by a maximum per unit cost, products must meet minimum standards of quality and reliability, the organization must function at a minimum rate of efficiency, and so on. How can these organizational goals be formulated and evaluated? Assessing the ability of an organization to meet a set of goals may require simulating the organization and observing its behavior as it processes its tasks. How is this to be done?

Environment Model

The design of an organization that is able to meet its organizational goals requires information about the environment in which it will function. The environment is the ultimate source of the organization's inputs and the destination of its products. The model needs to include knowledge about the rate at which its inputs will arrive and the variability of that rate, the characteristics of its inputs and their variability, interactions or correlations between inputs, the effects of outputs on inputs, and the degree to which it is ignorant of any of these things. The model is a prediction of what the environment will be like when the organization is functioning within it. How can this knowledge be represented?

Integration of Knowledge. How can the knowledge about the organization's task, its goals, and its environment be combined and used effectively when making choices during instantiation?

3.5.4 Summary

We have suggested that descriptions of organizational structure are important for the instantiation and maintenance of distributed systems over large heterogeneous networks. Current languages for describing organizational structure do not allow descriptions of arbitrary relations and are incapable of describing higher-order relations. We have identified three types of constraints (restriction, group, and preference) and have used them to describe and instantiate arbitrary and complex organizational relations. We have provided an algorithm for finding solutions to interacting constraints employed in descriptions of relations. Finally, we have tested these techniques by incorporating them in an organizational instantiation language, called EFIGE, and its interpreter.

Appendix A: The Complete Constraint Solution Algorithm

```
begin
Order relations with RELATION-TYPE "new" by EVALUATION-ORDER.
for
    each relation with RELATION-TYPE "new"
do
    Determine the members of each domain of the relation.
    for
        all members in the relation
    do
        Apply appropriate RESTRICTION constraint to members of each domain
            to form CANDIDATES set.
    end-for.
    for
        all members in the relation
    do
        Make CANDIDATE sets mutually consistent.
    end-for.
    if
        any member is left with an empty CANDIDATES set
    then
        Indicate over-constrained.
    else
        for
            all members in the relation
        do
            Apply appropriate GROUP constraint to each domain's CANDIDATES
                sets to form GROUP sets for each domain.
        end-for.
    end-if


    Set PROCESSED stack to empty.
```

```
Set UNSOLVED list to list of all members in the relation.
repeat
    while
        (not over-constrained) and (UNSOLVED list not empty)
    do
        Set CURRENT-MEMBER to member in UNSOLVED list with smallest
            product of the number of GROUP sets for each domain.
        Set REJECTED list of CURRENT-MEMBER to empty.
        repeat
            if
                GROUP set for any domain of CURRENT-MEMBER is empty
            then
                Add members in REJECTED list to GROUP set.
                Set REJECTED list to empty.
                if
                    PROCESSED stack is empty
                then
                    Indicate over-constrained.
                else
                    Set CURRENT-MEMBER to top of PROCESSED stack.
                    Pop top of PROCESSED stack.
                end-if


            else
                Use PREFERENCE constraints to select a group for each
                    domain from GROUP sets of CURRENT-MEMBER.
                Set SOLUTION of CURRENT-MEMBER to selected groups.
                Delete selected groups from GROUP sets of CURRENT-MEMBER.
                Make GROUP sets of members in UNSOLVED list consistent
                    with SOLUTION of CURRENT-MEMBER.
                if
                    no member in UNSOLVED list left with an empty GROUP set
                then
                    Delete CURRENT-MEMBER from UNSOLVED list.
                    Add CURRENT-MEMBER to PROCESSED list.
                    Indicate local-success.
```

```
                end-if
            end-if

            if
                (not over-constrained) and (not local-success)
            then
                Undo consistency changes to members in UNSOLVED list.
                Add SOLUTION of CURRENT-MEMBER to REJECTED list
                    of CURRENT-MEMBER.
            end-if
        until (local-success) or (over-constrained).
    end-while
    if
        (additional-solutions-requested) and (not over-constrained)
    then
        for
            all members in the relation
        do
            Save SOLUTION of member.
        end-for.
        Set CURRENT-MEMBER to top of PROCESSED stack.
        Pop top of PROCESSED stack.
        Add SOLUTION of CURRENT-MEMBER to REJECTED list
            of CURRENT-MEMBER.
    end-if
  until (no additional-solutions-requested) or (over-constrained).
end-for
end
```

Appendix B: Domain Specific Functions

This appendix describes some functions needed for the description of organizations in the DVMT. A few of these have been seen in the examples throughout this report.

The simplest class of functions tests, collects, or summarizes the contents of particular fields in a given description. For example, one function in this class tests if a node's knowledge sources include a particular set of knowledge sources, and another determines if a sensor is capable of detecting vehicular activity in some portion of a given region. Other examples examine a subfield of the CHARACTERISTICS field in a sensor description to see if it includes a given list of values, compare the rating of a sensor's accuracy at classifying and locating signals with a given value, test to see if an organization has interest-areas that intersect with a supplied list of interest-areas, list the classes of signals a sensor detects, and determine the communication knowledge sources an individual will need based on its communication activity.

Another function class tests for the presence of a relation between fields in multiple descriptions or summarizes data from multiple descriptions. The example in Section 3.2 includes a function from this class which finds all of the combinations of sensors (from a list of sensors) that will, between them, scan all of a given region. Other functions from this class check that the distance between two locations does not exceed a given value, order individual organizations by their distance from a given location and return a list of the n closest organizations, and return a region that encloses all of the regions scanned by a list of sensors.

A number of functions were written to perform operations on *regions*, two-dimensional rectangular areas specified by the coordinates of their lower-left and upper-right corners. (Technically, a region is just one dimension of an interest-area and it is only a matter of convenience that they are all rectangles in the DVMT.) Some of these functions

- compute a minimum enclosing rectangle
- accept a list of rectangles and return the rectangle that is overlapped by all of them
- accept two overlapping rectangles, break up the area of the first rectangle that is not overlapped by the second into smaller rectangles (at most, three are required), and return them in a list
- fill a rectangle with overlapping rectangles (this is used by a composite structure to assign regions to its components).

A final function, with arguments an integer, a maximum divisor, and a minimum quotient, returns a list of no more than maximum divisor integers, all of them at least as large as the minimum quotient, such that they all add up to the original number. This function is used to distribute employees to subhierarchies (in the example of Section 3.3), where managers do not want to manage more than some number of subhierarchies (maximum divisor) and a minimum number of employees are required to make up a hierarchy (minimum quotient).

Acknowledgements

This research was sponsored, in part, by the National Science Foundation under grants MCS-8306327 and DCR-8500332, and by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract NR049-041.

References

[1] Lars Warren Ericson, "DPL-82: A Language for Distributed Processing," *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pp. 526–531, Miami/Ft. Lauderdale, Florida, October 1982.

[2] Willie Y-P. Lim. "HISDL—A Structure Description Language," *Communications of the ACM*, vol. 25, no. 11, pp. 823–830, November 1982.

[3] Mark S. Fox, *Organization Structuring: Designing large complex software*, Technical Report 79-155, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, December 1979.

[4] Victor R. Lesser, Daniel Serrain, and Jeff Bonar, "PCL: A process-oriented job control language," *Proceedings of the First International Conference on Distributed Computing Systems*, pp. 315–329, 1979.

[5] Richard J. LeBlanc and Arthur B. Maccabe, "The Design of a Programming Language Based on Connectivity Networks," *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pp. 532–541, Miami/Ft. Lauderdale, Florida, October 1982.

[6] Anita K. Jones and Karsten Schwans, "TASK Forces: Distributed software for solving problems of substantial size," *4th International Conference on Software Engineering*, pp. 315–330, Munich, Germany, September 1979.

[7] David Waltz, "Understanding Line Drawings of Scenes with Shadows," in Patrick Winston, editor, *The Psychology of Computer Vision*, pp. 19–91, McGraw-Hill, 1975.

[8] Reid G. Smith, "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver," *IEEE Transactions on Computers*, vol. C-29,, no. 12, pp. 1104–1113, December 1980.

[9] Jasmina Pavlin, "Predicting the Performance of Distributed Knowledge-Based Systems: A Modeling Approach," *Proceedings of the National Conference on Artificial Intelligence*, pp. 314–319, Washington, D.C., August 1983.

[10] Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill, "Coherent Cooperation Among Communicating Problem Solvers," *IEEE Transactions on Computers*, 1987. (Accepted for publication. Also published as Technical Report 85-15, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, April 1985).

[11] Scott Reed and Victor R. Lesser, *Divison of Labor in Honey Bees and Distributed Focus of Attention*, Technical Report 80-17, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, September 1980.

[12] Daniel D. Corkill, *A Framework for Organizational Self-Design in Distributed Problem Solving Networks*, Ph.D. Thesis, University of Massachusetts, February 1983. (Also published as Technical Report 82-33, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, December 1982.).

[13] Mark Stefik, "Planning with Constraints (MOLGEN: Part 1)," *Artificial Intelligence*, vol. 16, pp. 111–140, 1981.

[14] David Stemple and Tim Sheard, "Specification and Verification of Abstract Database Types," *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 248–257, Waterloo, Ontario, Academic Press 1984.

[15] Victor R. Lesser and Daniel D. Corkill, "The Distributed Vehicle Monitoring Testbed: A tool for investigating distributed problem solving networks," *AI Magazine*, vol. 4, no. 3, pp. 15–33, Fall 1983.

[16] Mark S. Fox, *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*, Ph.D. Thesis, Carnegie-Mellon University, 1983, (Also published as Technical Report CMU-CS-83-161, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213).

H. Edward Pattison, Daniel D. Corkill and Victor R. Lesser
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003