Instantiating Descriptions of Organizational Structures

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

COINS Technical Report 85-45
November 1985

Department of Computer and Information Sciences
University of Massachusetts
Amherst, Massachusetts 01003

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.

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 domain specific and complex relations found in many organizations. EFIGE is a language that allows such relations to be specified. The language 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 paper describes EFIGE and shows how relations refined with constraints may be implemented.

Contents

Chapter 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 include DPL-82 [Ericson 1982], HISDL [Lim 1982], ODL [Fox 1979], PCL [Lesser, Serrain, & Bonar 1979], PRONET [LeBlanc & Macabbe 1982], and TASK [Jones & Schwans 1979]. These languages, however, 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 (e.g., 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). 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 design 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 the ability 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 *refine* a relation because they "reduce" the number of possible pairings of a member of one domain with the members of another. More precisely, a relation defines a set of ordered pairs (in general, n-tuples) that is the cartesian cross-product of each domain of the relation, while a constraint is a predicate that selects some of the pairings as more significant than others.

The introduction of 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, complicates organization instantiation. To instantiate a relation, solutions must be found for each of the constraints with which it was refined. 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 AI literature that is used to eliminate inconsistent assignments of values to constraints [Waltz 1975]. This approach is limited, however, because it tries to choose solutions for 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 chapter we present an example of an organizational structure, then discuss its description and requirements for an organizational description language. Chapter 3 indicates how structures are described within our framework, Chapter 4 describes how descriptions are instantiated, and the last chapter discusses both the current status of our work and future research.

Chapter 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 2.1 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 2.2 shows another instance of the hierarchical organizational structure. It has five integrating nodes, sixteen synthesizing nodes, and sixteen sensing nodes.

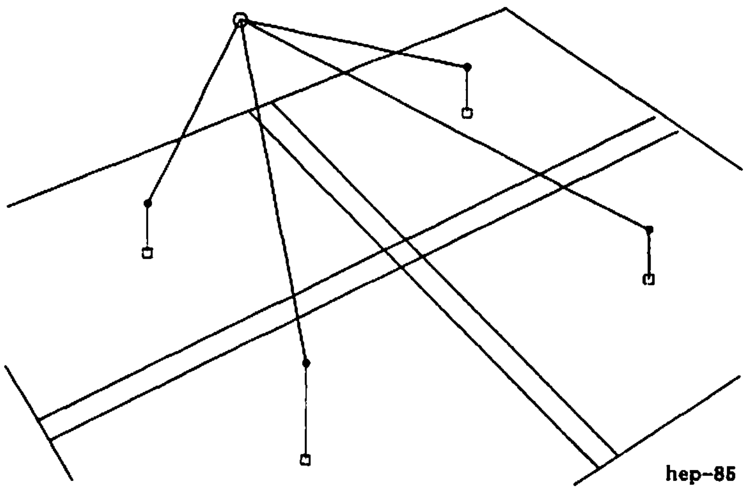


Figure 2.1: *An instance of the hierarchical organizational structure with one integrating node (circle), four synthesizing node (dots), and four sensing nodes (squares).*

Figures 2.1 and 2.2 show two *instances* of the same organizational *class*. The goal of our work has been 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 (here: sensing, synthesizing, and integrating nodes) and the relations between them (communication relations are emphasized in the figures: sensing and synthesizing nodes, synthesizing and integrating nodes, 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 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

[1]This last relation is not instantiated in Figure 2.1 because there is only one integrating node.

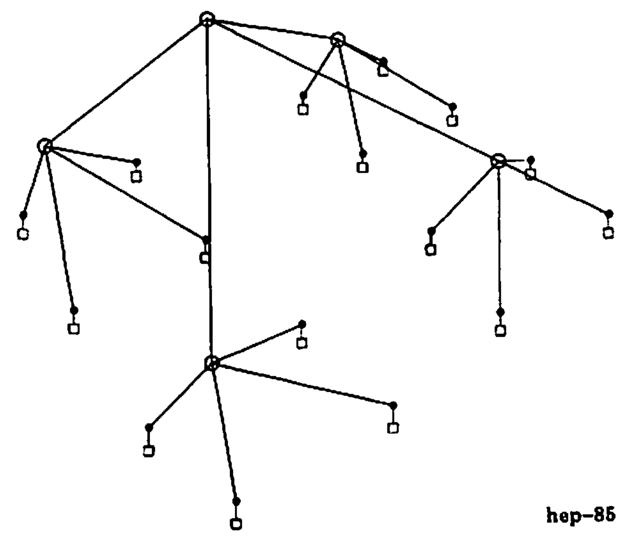


Figure 2.2: *The hierarchical organizational structure with five integrating nodes, sixteen synthesizing nodes, and sixteen sensing nodes.*

receive signal information from sensing nodes. For that reason, their descriptions must also be independent of details specific to single instances of the organization.

We have said that the key features of an organizational class are its components and the relations between them. In the rest of this chapter, we try to identify what information a description of these organizational features will need to include and the range of values that will have to be accommodated. We start, however, with a discussion of the organization's purpose.

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 estimating 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). Further, distributed systems, in general, are expected to be robust: able to adjust to node failures and to have performance degrade gracefully as error in the system increases.

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 sub-set 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 sub-regions 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 (e.g., 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 (e.g., 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, the 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. There seem to be 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 contacts.

Knowledge. There seem to be 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 the 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, say, 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 Sub-Contractors. 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 sub-problems [Smith 1980]. This method does not rely on knowing in advance who is capable of solving the problems or sub-problems, 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 isn't 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 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 [Pavlin 1983].

2..3 Relations Between Components

Components in an organization do not exist—nor do they function—independent of one another. Components interact. Commands, information, and sub-assemblies (or partial solutions) are passed between them; 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 a single relation will be required to exist between just two components. It is more likely that a conjunction of relations will occur and that 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 2.1 and 2.2: each inter-node line represents an instance of a communication relation. The communication relation is used to identify a component's sources of one especially valuable resource—information—and to identify the consumers of the information it produces.

Equally important is exactly 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 another 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 [Durfee, Lesser, & Corkill 1985].

Authority

Authority is a relation that modifies another: the communication relation. It 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 want to allow it 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 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 [Reed & Lesser 1980]; Corkill has experimented with the affects of skeptical nodes in distributed problem solving organizations performing signal interpretation [Corkill 1982].

In general, it seems that relations can often be described on two levels. The first level is a (relatively) global level that outlines the relation and its participants. The second is the local level, in which details and individual variance are elaborated.

Location, Proximity, etc.

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.

2..4 Composite Components

Organizations are often composed of sub-organizations. In order to simplify descriptions of such organizations, the sub-organizations are treated as single components and the interactions among these components are detailed; then the components are "enlarged" to reveal the sub-organization they represent. While these *composite* components don't 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 "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 2.2) is instantiated as an integrating node with hierarchical organizations as its components. Each of these hierarchical sub-organizations 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 virtual components need to be created: the synthesizing nodes are created right away. This is the case for five nodes, for example.

Chapter 3

Describing Organizational Structures with EFIGE

This chapter introduces a language, called EFIGE (pronounced "effigy"), for describing organizational structures. Descriptions of organizations in EFIGE are hierarchical. That is, they may be of either individual or of composite structures, and a composite structure's components may be individual or composite. Figure 3.1 shows part of the description of the hierarchical organization presented in Figures 2.1 and 2.2 (a complete description of the hierarchical organizational structure is given in Appendix A). 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 recursively defined. This is the case with the hierarchical organization.

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, below).*

```
     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 *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)
     )
     (COMPONENT-NAME sub-hierarchy
```

The component, `sub-hierarchy`*, is described by the description,* `hierarchical`*, thus this component is recursive.*

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

Figure 3.1: *Part of the description of the* `hierarchical` *organization.*

Figure 3.2 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.

```
(NAME synthesizing-node
```

This description is of an individual structure, it has no components.

```
 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-region` *was bound to a description of a problem-solving region in the* LOCAL-VALUES *field.*

```
 RESPONSIBILTIES
   ((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-kses ?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 used to simulate the individual's behaviour.*

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

Figure 3.2: *Part of the description of an individual.*

3.1 Relations

The hierarchical method 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 structures *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 just 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 3.3). A declaration merely specifies that a relation exists between one, or more, components. The composite structure in which the declaration is made does not know if a component is an individual or composite, but it does not need to. Either type of component 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 3.4). 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 3.5). 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 3.3 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 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 3.3 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 3.4 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

```
RELATIONS
```

Entries in the RELATIONS *field are given names.*

```
  ((RELATION-NAME      sensor-synthesizer
```

Entries of type new *are used to declare the existance 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 3.3: *Example of the declaration of a relation.*

which includes `middle-integrator`.

```
(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 structures 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 3.4: *Example of the forwarding of a relation.*

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 3.5 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` (Figure 3.1). It refines the relation `sensor-synthesizer`, which referred to it in Figure 3.3. Since this is (implicitly) a communication relation, `to-sensor` includes information that is to be associated with the relation (e.g., the direction messages are to travel in the relation, their format, communication strategies, etc.).

```
(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 3.6).*

```
 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 its format.*

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

Figure 3.5: *Example of relation refinement.*

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 $\{x | (x \in F) \wedge (Fx)\}$,

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 [Jones & Schwans 1979]. 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 with a large local memory. Artificial Intelligence (AI) programs that perform planning tasks also use restriction constraints. MOLGEN, when planning experiments in molecular genetics, will generate, for example, a constraint restricting the choice of a bacterium to one that resists an antibiotic [Stefik 1981].

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 sub-set, 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 [Stemple & Sheard 1984].

Preference. Preference constraints implicitly define a partial order over a set (a total order if execution is deterministic) 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

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

and

$$V_1 = X.$$

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 $(S^k X) \equiv (S V_k)$. 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 is the familiar 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:

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

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 it 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:

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

where

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

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

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

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

In actuality, union is too simple a function 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 (jl), then the constraints for l should include that tuple in their selection as well. If this isn't 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). Chapter 4 presents an algorithm for evaluating constraints and combining them in such a way that they are consistent.

Figure 3.6 shows 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 Figure 3.6, it tests that the information associated with the relation (in the ADDITIONAL-DATA field, see Figure 3.5) 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.

```
    CONSTRAINT
```

The PARTNERS *field lists the* other *domains of the relation: not the one to which the owner of this constraint belongs. A constraint of each type will have to be provided for each of the domains listed.*

```
        (PARTNERS       (sensor)
```

The predicates in the RESTRICTIONS *field act as filters, rejecting members of the other domains that don't meet their criteria.*

```
        RESTRICTIONS
```

The predicate `compatable-communication?` *will examine the descriptions in the* ADDITIONAL-DATA *fields of this relation (see Figure 3.5) and that of each member of the* `sensor` *domain for consistency (e.g., since the* DIRECTIONS *field in this relation says* `receive`*, the other must say* `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.*

```
        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 3.6: *Constraints.*

3.3 The Procedural/Declarative Interface

Figure 3.6 illustrates 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 3.6, the function, `sensors-that-cover-rectangle`, 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 many other applications. The declarative part of EFIGE, the 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.

Chapter 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 with which it was refined, but this is complicated 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 nodes. In this chapter, we first present the algorithm for finding solutions to constraints, then briefly describe how the `hierarchical` organization is instantiated.

4..1 Constraint Solution Algorithm

The algorithm we use for finding solutions to the interacting constraints that refine 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; from them the global solution will be built. 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 don't 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 or 2) not contain the name of the member just processed, if the solution doesn't 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 follows.

```
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.
```

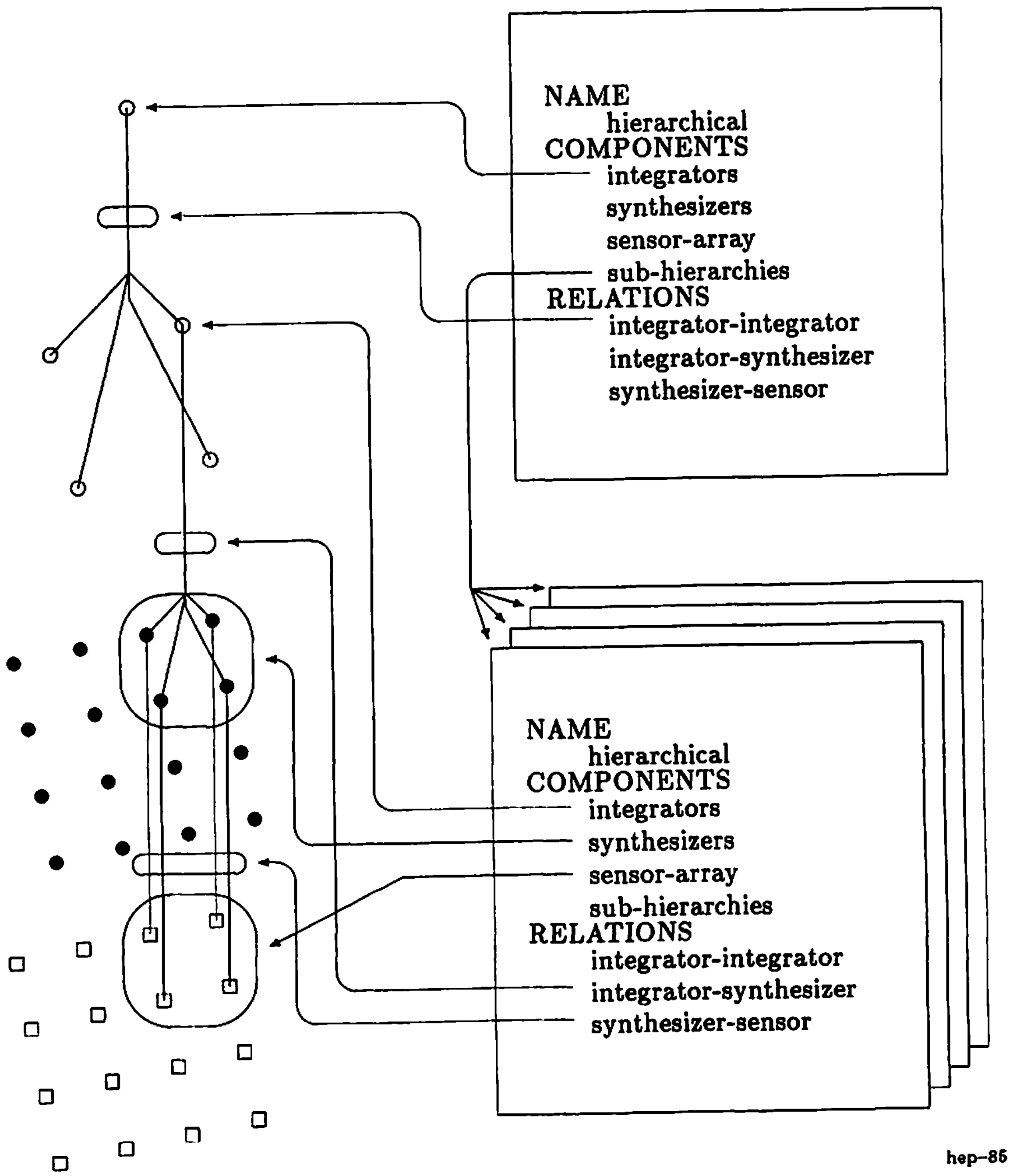
```
                    Indicate local-success.
                end-if.
            end-if.
            if
                (not over-constrained) and (not local-success)
            then
                Undo consistency changes to members in UNSOLVED list.
                Add SOLUTION of CURRENT-MEMBER 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.
```

4..2 Instantiation of the Hierarchical Organization

Figure 4.1 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. (See Appendix A for the complete description of the hierarchical organization.) When the upper hierarchical structure was instantiated, the preconditions of only two of its components evaluated to true: the integrators component and the sub-hierarchies component. One copy of the integrators, and four of the sub-hierarchies, were instantiated. The integrator-integrator relation is implemented because, at this point, it is actually an integrator-sub-hierarchies relation. In the sub-hierarhcies, membership in the relation is forwarded to their integrators components.

Each of the sub-hierarchies components is another hierarchical organization. This time, however, in each of them the precondition for the sub-hierarchies 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, a integrator-synthesizer relation and a synthesizer-sensor relation is implemented.

Figure 4.1: (Next Page)*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.*


NAME
hierarchical
COMPONENTS
integrators
synthesizers
sensor-array
sub-hierarchies
RELATIONS
integrator-integrator
integrator-synthesizer
synthesizer-sensor
NAME
hierarchical
COMPONENTS
integrators
synthesizers
sensor-array
sub-hierarchies
RELATIONS
integrator-integrator
integrator-synthesizer
synthesizer-sensor
hep-85

Chapter 5

Status and Ongoing Research

EFIGE has been implemented in VAX/LISP, Digital Equipment Corporation's implementation of Common Lisp. Descriptions have been written of organizational structures for use in the Distributed Vehicle Monitoring Testbed (DVMT) [Lesser & Corkill 1983].(The hierarchical organization used as an example in this paper 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.

5.1 Organizational Self-Design

The long-term goal of this research is organizational self-design. An organization with this ability will perform the following tasks: 0) monitor the organizational structure's effectiveness in directing organizational activities, 1) identify new organizational structures appropriate to a new situation, 2) select the best among them, and 3) 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 1 and 2 of the organizational self-design task.

Repairing Broken Organizations Another simplification of organizational self-design 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 (roughly step 0), but the emphasis is then placed on recovering lost functionality without adopting a new organizational structure (eliminating steps 1 and 2, simplifying step 3). 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 organization 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 [Fox 1979]. 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 [Pavlin 1983].

Organizational Goals The goals of an organization are its desired performance abilities: e.g., 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 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?

5.2 Improvements to EFIGE

Investigations directed toward finding answers to three questions should result in an improved system. These questions are: how can the constraint mechanism be made more general, how can search efficiency be improved, and 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. It could be left to a distributed operating system to create a "virtual" processing network that matched the one assumed by the interpreter, but we have chosen to investigate how to describe *bottom-level* constraints and how to incorporate them 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 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 [Stefik 1981]. 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 the more complex constraints used in our work, 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, 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 [Fox 1983]. 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. Investigation of the assignment of credit problem in machine learning, and on the problem of resolving deadlock during resource allocation in operating systems may also provide clues on how to locate conflicting constraints.

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?

5.3 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 and presented a method that uses them to describe and instantiate arbitrary and complex 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 a language, EFIGE, and its interpreter.

Bibliography

[Corkill 1982] Daniel D. Corkill. *A Framework for Organizational Self-Design in Distributed Problem Solving Networks.* Ph.D. Thesis, University of Massachusetts, 1983. Available as COINS Technical Report 82-33, Computer and Information Sciences Department, University of Massachusetts, Amherst, MA 01003.

[Durfee, Lesser, & Corkill 1985] Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill. *Coherent Cooperation Among Communicating Problem Solvers.* COINS Technical Report 85-15, Computer and Information Sciences Department, University of Massachusetts, Amherst, MA 01003.

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

[Fox 1979] Mark S. Fox. *Organization Structuring: Designing large complex software.* Technical report CMU-CS-79-155, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

[Fox 1983] Mark S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* PH.D. Thesis, Carnegie-Mellon University, 1983. Available as CMU-CS-83-161, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213.

[Jones & Schwans 1979] Anita K. Jones and Karsten Schwans. TASK Forces: Distributed software for solving problems of substantial size. *4th International*

Conference on Software Engineering (Proceedings), Munich, Germany, September 1979, pp 315–330. IEEE.

[LeBlanc & Macabbe 1982] 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,* Miami/Ft. Lauderdale, Florida, October 1982, pp 532–541. IEEE.

[Lesser & Corkill 1983] Victor R. Lesser and Daniel D. Corkill. The Distributed Vehicle Monitoring Testbed: A tool for investigating distributed problem solving networks. *AI Magazine,* Fall 1983, 4(3), pp 15–33.

[Lesser, Serrain, & Bonar 1979] 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,* 1979, pp 315–329. IEEE.

[Lim 1982] Willie Y-P. Lim. HISDL—A Structure Description Language. *Communications of the ACM,* November 1982, 25(11), pp 823–830.

[Pavlin 1983] Jasmina Pavlin. Predicting the Performance of Distributed Knowledge-Based Systems: A Modeling Approach. *AAAI-83 Proceedings,* William Kaufman, Inc. Los Altos, CA 94022.

[Reed & Lesser 1980] Scott Reed and Victor R. Lesser. *Division of Labor in Honey Bees and Distributed Focus of Attention.* COINS Technical Report 80-17. Computer and Information Sciences Department, University of Massachusetts, Amherst, MA 01003.

[Smith 1980] Reid Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers,* December 1980, C-29(12), pp 1104–1113.

[Stefik 1981] Mark Stefik. Planning with Constraints (MOLGEN: Part 1). *Artificial Intelligence,* 1981, 16, pp 111–139.

[Stemple & Sheard 1984] 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,* Waterloo, Ontario, April 1984, pp 248–257.

[Waltz 1975] David Waltz. Understanding Line Drawings of Scenes with Shadows. In Patrick Winston (editor), *The Psychology of Computer Vision,* McGraw-Hill, New York, NY.

Appendix A

Complete Description of the Hierarchical Organization

This appendix contains all of the descriptions needed for the hierarchical organization, which includes the composite description, hierarchical, and the descriptions of the individual components integrating-node, synthesizing-node, and vanilla-sensor. The descriptions have been annotated in order to provide explanations of the syntactic constructs. The first description is of the hierarchical organization.

```
;;;; ************************************************************************
```

All descriptions are given names.

```
(NAME  hierarchical
```

A composite *description has components.*

```
 TYPE  composite
```

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

```
 PARAMETERS
```

`Number-of-nodes` *is the total number of synthesizing and integrating nodes to be used in the instantiation of the organization.* `Number-of-sensors` *is the total number of sensing nodes to be used.*

```
   ((PARAMETER-NAME number-of-nodes     DEFAULT 21)
    (PARAMETER-NAME number-of-sensors   DEFAULT 15)
```

Areas in the DVMT *are given in terms of a two dimensional coordinate system.* `region` *is the entire area that is to be monitored and is specified as a rectangle with minimum x and y coordinate values of* 0 *and maximum x and y values of* 22.

```
    (PARAMETER-NAME region              DEFAULT '(0 0 22 22))
```

The distance the areas scanned by sensing nodes should overlap.

```
    (PARAMETER-NAME sensor-overlap      DEFAULT 2)
   )
```

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

```
 LOCAL-VALUES
   ((one-less-node              (1- number-of-nodes))
```

A `group-divide` *of n, l, and m returns a list of no more than l integers, none of which is less than m, and that sum to n.*

```
   (list-of-hierarchy-sizes  (group-divide one-less-node 4 3))
   (number-of-hierarchies    (length list-of-hierarchy-sizes))
   (list-of-hierarchy-sensor-counts
              (group-divide
                  number-of-sensors  number-of-hierarchies  1))
  )
```

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

```
 COMPONENTS
```

Components are given local names.

```
   ((COMPONENT-NAME integrators
```

The organizational structure of the component is given in the description named `integrating-node`. `Region-i` *is a parameter of* `integrating-node`. *It will be given the value of* `region`, *one of the parameters to this description.*

```
      DESCRIPTION    (integrating-node  ((region-i 'region)))
```

The organization will be instantiated once as this component.

```
      COPIES          (1)
```

Components will be instantiated only if the entry in the PRECONDITION *field evaluates to* `true`*—in this case, when the value given the parameter,* `number-of-nodes`, *is greater than 2.*

```
      PRECONDITION   (> number-of-nodes 2)
     )
     (COMPONENT-NAME synthesizers
```

The component, synthesizers, *is described by the organizational description,* synthesizing-node. Synthesizing-node *has a parameter,* region-s, *which will be set to the value of* worker-region, *defined in the* COPIES *field below.*

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

Synthesizing-node *is to be instantiated* one-less-node *times (as the component,* synthesizers*; it could be instantiated for another component, as well).* 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 assigning a sequence of values to variables. Here, the variable,* worker-region, *is to receive* one-less-node *values, each of which will be used in a different instantiation of* synthesizing-node.

```
        (VARY
           (worker-region
```

Because parameter values are evaluated before substituting them for occurrences of the parameter's name in a description, regions need to be quoted since they are specified with a list.

```
                (quote-list-elements
```

The function, fill-rectangle-with-overlapping-rectangles, *will divide the rectangular area,* region, *into* one-less-node *sub-regions. These regions will overlap each other by a distance of* sensor-overlap *in both the x and y directions.*

```
                    (fill-rectangle-with-overlapping-rectangles
                        'region  one-less-node
                        (list sensor-overlap sensor-overlap))))
        ))
```

This component is to be instantiated when number-of-nodes *is within the range 3–5, inclusive.*

```
     PRECONDITION   (within-subrange? number-of-nodes 3 5)
    )
    (COMPONENT-NAME sensor-array
     DESCRIPTION    (vanilla-sensor ((region-v sensor-region)))
     COPIES         (number-of-sensors
        (VARY
           (sensor-region
               (quote-list-elements
                   (fill-rectangle-with-overlapping-rectangles
                       'region  number-of-sensors
                       (list sensor-overlap sensor-overlap))))
        ))
```

The sensor-array *component should be instantiated when the* synthesizers *component is and if* number-of-sensors *is greater than zero.*

```
     PRECONDITION   (and (within-subrange? number-of-nodes 3 5)
                         (> number-of-sensors 0))
    )
    (COMPONENT-NAME sub-hierarchy
```

The component, sub-hierarchy, *is described by the description,* hierarchical, *thus this component is recursive. Three of the parameters will be given new values:* region, number-of-nodes, *and* number-of-sensors.

```
     DESCRIPTION    (hierarchical
                       ((region             one-region)
                        (number-of-nodes    hierarchy-node-count)
                        (number-of-sensors  hierarchy-sensor-count)
                        (sensor-overlap     sensor-overlap)
```

```
                    ))
      COPIES          (number-of-hierarchies
         (VARY
           (one-region
               (quote-list-elements
                   (fill-rectangle-with-overlapping-rectangles
                       'region  number-of-hierarchies
                       (list sensor-overlap sensor-overlap))))
```

`list-of-hierarchy-sizes` *and* `list-of-hierarchy-sensor-counts` *were defined in the* LOCAL-VALUES *field. They are lists of integers.*

```
          (hierarchy-node-count     'list-of-hierarchy-sizes)
          (hierarchy-sensor-count   'list-of-hierarchy-sensor-counts)
        ))
```

`Sub-hierarchy` *is instantiated only if* `number-of-nodes` *is greater than five; when this is the case, the only other component that will be instantiated is* `integrators`.

```
      PRECONDITION   (> number-of-nodes 5)
    ))
```

The RELATIONS *field defines relations between components.*

```
 RELATIONS
```

This first relation is for communication between sensing nodes and synthesizing nodes.

Relations are given names.

```
    ((RELATION-NAME     sensor-synthesizer
```

A RELATION-TYPE of new *indicates that this entry will declare the existence of a relation between one or more groups of components. (Other types of entries are* `forward` *and* `refine`*; they will be described as they occur.)*

```
    RELATION-TYPE     new
```

Implementation of one relation may depend on having another relation already implemented. The EVALUATION-ORDER entry specifies in what order new relations should be implemented.

```
    EVALUATION-ORDER  1
    RELATE
```

Each domain of a relation is given a name; here they are sensor *and* node. *The members of a domain are specified by giving a component name combined with a RELATION-NAME from the organizational description of the component. The two names are separated by a dollar sign.*

```
            ((sensor  sensor-array$to-synthesizer)
             (node    synthesizers$sensor-data)
            )
    )
```

This relation is for communication between synthesizing nodes and integrating nodes.

```
    (RELATION-NAME     synthesizer-integrator
     RELATION-TYPE     new
     EVALUATION-ORDER  2
     RELATE
            ((integrator   integrators$lower-exchange)
             (subordinate  synthesizers$to-integrator)
            )
    )
```

This relation is for communication between high-level integrating nodes and low-level integrating nodes.

```
    (RELATION-NAME     integrator-subhierarchy
     RELATION-TYPE     new
     EVALUATION-ORDER  3
     RELATE
            ((integrator   integrators$lower-exchange)
             (subordinate  sub-hierarchy$middle-integrator)
            )
    )
    (RELATION-NAME     middle-integrator
```

An entry with type `forward` *is used in the description of a composite organization to confer its own membership in a relation upon one of its components. Here, the assumption is made that—somewhere—the organization,* `hierarchical`, *is a component of another organization and within the description of that organization's relations is a reference to* `middle-integrator`. *(In fact, since the description of* `hierarchical` *is recursive, that other organization is itself and the reference is in the declaration of the relation,* `integrator-subhierarchy`, *above.)*

```
     RELATION-TYPE     forward
```

Pass membership in the relation to the entry in the RELATIONS *field of the organizational description for the the component,* `integrators`, *with the name,* `upper-exchange`.

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

); end "hierarchical".

;;;; ***************************************************************************
```

The remaining descriptions are of individual structures. The first of these is called `integrating-node`. It appears in the description of the `hierarchical` organization as the organization to be instantiated for the `integrators` component.

```
;;;; ****************************************************************************

(NAME  integrating-node
```

For descriptions of organizations in the DVMT, we distinguish between two types of individual structures: nodes and sensors. `Sensor` *designates a sensing nodes, while* `node` *is used for any other kind of node.*

```
 TYPE  node

 PARAMETERS
   ((PARAMETER-NAME region-i  DEFAULT '(0 0 22 22))
   )

 LOCAL-VALUES
```

These two LOCAL-VALUES *variables are set to descriptions of interest areas. The interest areas describe the boundaries of a region in the four dimensional space in which problem solving is performed. They differ only along the dimension that is the level of abstraction at which problem solving is performed. They both cover the same area through which vehicles move (the actual value depending on the value available for the parameter,* `region-i`*), extend for all time periods, and include all event classes.*

```
   ((vt-everywhere
        '(ABSTRACTION-LEVELS '(vt)
          INTEREST-REGION    'region-i
          TIME-PERIODS       *all
          EVENT-CLASSES      *all
```

```
      ))
   (pt-everywhere
      '(ABSTRACTION-LEVELS '(pt)
        INTEREST-REGION    'region-i
        TIME-PERIODS       *all
        EVENT-CLASSES      *all
       ))
  )
```

RESPONSIBILITIES

Integrating nodes will be responsible for problem solving in the interest regions described by `vt-everywhere` *and by* `pt-everywhere`*. They should, however, allocate most of their time to processing at the* `pt` *abstraction level.*

```
  ((PROCESS-AREA  '(vt-everywhere)
    IMPORTANCE    *moderate*
   )
   (PROCESS-AREA  '(pt-everywhere)
    IMPORTANCE    *extremely-high*
  ))
```

```
RESOURCES
   (KNOWLEDGE-SOURCES
```

Integrating nodes will need knowledge about problem solving in their particular region of the problem-solving space.

```
      ((KS-NAMES        (union  *vehicle-merge-connect  *vehicle-to-pattern
                                *pattern-processing)
        GOODNESS        *extremely-high*
        RESOLVING-POWER *zilch*
        RUNTIME         '(0 1)
      )
```

Integrating nodes will need to know how to communicate with other nodes in the network.

```
        (KS-NAMES          (determine-communication-kses ?this-structure)
         GOODNESS          *extremely-high*
         RESOLVING-POWER   *zilch*
         RUNTIME           '(0 0)
        ))
```

Meta-level knowledge expressed in an arcane format intelligible only to the DVMT.

```
      SUBGOALING
        '(((pt) (((vt) ((1 10000)))))
         )
     )

   CHARACTERISTICS
```

The node is assigned a location within the region through which the vehicles move.

```
     (LOCATION    (rectangle-center region-i)
```

The node's speed expressed in a form understood by the DVMT.

```
      NODE-SPEED  nil
     )

   RELATIONS
```

This entry refines the communication relation between upper-level integrating nodes and lower-level integrating nodes from the lower-level node's perspective. That is, when a node that has been instantiated from this description is the low-level node in an upper-level lower-level communication relation, this entry is used to refine the relation.

```
((RELATION-NAME  upper-exchange
  RELATION-TYPE  refine
  CONSTRAINT
```

An n-ary relation has n domains, each of which is given a name. The PARTNERS *field lists these names, except for the name of the domain in which this node occurs. A constraint must be provided for each of the relation's other domains and this list helps the user to remember what those domains are and insures consistency between the relations declaration and its refinement.*

```
      (PARTNERS      (integrator)
```

A restriction constraint must be provided for each of the domains listed in the PARTNERS *field. This constraint tests the consistency between the information supplied in the* ADDITIONAL-DATA *field of this relation and that supplied by the* ADDITIONAL-DATA *field of each member of the* `integrator` *domain of the relation.*

```
       RESTRICTIONS
            ((integrator  (compatible-communication?
                               ?this-relation  ?partner-relation))
            )
```

Those members of the relation's `integrator` *domain that satisfied the restriction constraint, above, are called candidates and are accessed with the symbol* `?candidates`. *This group constraint creates a single group from the candidates without eliminating any of them.*

```
       GROUPS
            ((integrator  (all-candidates ?candidates))
            )
```

The list of groups formed by the group constraint can be accessed with the symbol `?groups`. *This preference constraint simply takes the first group from the list (the group constraint above will only generate one group).*

```
PREFERENCE
      ((integrator  (first ?groups))
      )
)
```

The ADDITIONAL-DATA *field is used to add fields to a description; the presumption is that such additions will only need to be made when the described entity is a participant in a relation.*

```
ADDITIONAL-DATA
  ((communication
```

These entries describe the nature of the communication between low-level and high-level integrating nodes from the perspective of the low-level node. The low-level node sends hypotheses in the interest region defined by `vt-everywhere` *if their belief is greater than* 0. *It receives goals in the interest region,* `vt-everywhere`, *if their belief is greater than* 0.

```
((DIRECTION           send
  NATURE              (hyp)
  DISPATCHES
    ((CONDITION       (above-threshold 0)
      INFORMATION     (vt-everywhere)
    ))
 )
 (DIRECTION           receive
  NATURE              (goal)
  DISPATCHES
    ((CONDITION       (above-threshold 0)
      INFORMATION     (vt-everywhere)
```

```
              ))
           ))
        ))
     )
```

This entry is used to refine two relations: the integrator-synthesizer relation from the integrator perspective and the low-level high-level integrator relation from the perspective of the high-level node.

```
    (RELATION-NAME  lower-exchange
     RELATION-TYPE  refine
     CONSTRAINT
        (PARTNERS      (subordinate)
         RESTRICTIONS
              ((subordinate  (compatible-communication?
                                  ?this-relation  ?partner-relation))
              )
         GROUPS
              ((subordinate  (all-candidates ?candidates))
              )
         PREFERENCE
              ((subordinate  (first ?groups))
              )
        )
     ADDITIONAL-DATA
       ((communication
```

Note that this description of the communication is an inverse of the communication description for the `upper-exchange` *entry, above. For instance, there hypotheses are sent while here they are received. This is because the* `upper-exchange` *entry and the* `lower-exchange` *entry are used in different domains of the same relation.*

```
          ((DIRECTION           receive
            NATURE                (hyp)
            DISPATCHES
              ((CONDITION         (above-threshold 0)
                 INFORMATION      (vt-everywhere)
              ))
           )
           (DIRECTION             send
            NATURE                (goal)
            DISPATCHES
              ((CONDITION         (above-threshold 0)
                 INFORMATION      (vt-everywhere)
              ))
           ))
        ))
     ))

); end "integrating-node".

;;;; ***************************************************************************
```

This description is of synthesizing-node, which is the organization instantiated for the synthesizers component of the hierarchical organization.

```
;;;; ***************************************************************************

(NAME  synthesizing-node
```

Node is a type of individual structure used in the DVMT.

```
 TYPE  node

 PARAMETERS
```

```
  ((PARAMETER-NAME region-s  DEFAULT '(0 0 22 22))
  )

LOCAL-VALUES
  ((vt-sensor-regions
      '(ABSTRACTION-LEVEL   '(vt)
        INTEREST-REGION
```

The value of this INTEREST-REGION *field depends on the implementation of the* `sensor-synthesizer` *relation. The function* `relation-solution` *will search the* RELATIONS *field of* `?this-structure` *for an entry with* `sensor-data` *as its* RELATION-NAME *and return the group selected for the* `sensor` *domain when the relation was implemented. The function* `make-group-members-structures` *insures that* `sum-sensors-scan-regions` *will receive a list of descriptions of structures (as opposed to their relation entries that were used to refine the* `sensor-synthesizer` *relation, or the names of these entries).* `Sum-sensors-scan-regions` *returns the smallest rectangular area that contains all of the areas scanned by the sensing nodes in a group.*

```
             (sum-sensors-scan-regions
                (make-group-members-structures
                   (relation-solution 'sensor-data 'sensor ?this-structure)))
        TIME              *all
        EVENT-CLASS       *all
       ))
   (sl-sensor-regions
      '(ABSTRACTION-LEVEL   '(sl)
        INTEREST-REGION     'region-s
        TIME                *all
        EVENT-CLASS         *all
       ))
   (gl-sensor-regions
      '(ABSTRACTION-LEVEL   '(gl)
        INTEREST-REGION     'region-s
        TIME                *all
      EVENT-CLASS        *all
     ))
  (vl-sensor-regions
     '(ABSTRACTION-LEVEL  '(vl)
       INTEREST-REGION    'region-s
       TIME               *all
       EVENT-CLASS        *all
      ))
 )
```

RESPONSIBILITIES

Work in different interest regions has different priorities.

```
  ((PROCESS-AREA  '(sl-sensor-regions gl-sensor-regions)
    IMPORTANCE    *tiny*
   )
   (PROCESS-AREA  '(vl-sensor-regions)
    IMPORTANCE    *very-low*
   )
   (PROCESS-AREA  '(vt-sensor-regions)
    IMPORTANCE    *moderate*
  ))
```

RESOURCES

```
   (KNOWLEDGE-SOURCES
```

*The symbols beginning with * each represent a list of knowledge source names; they are all joined into one list.*

```
      ((KS-NAMES        (union
                            *vehicle-merge-connect *vehicle-join-extend-form
                            *goup-processing       *signal-processing
```

```
                              '(sensors))
         GOODNESS           *extremely-high*
         RESOLVING-POWER    *zilch*
         RUNTIME            '(0 1)
        )
        (KS-NAMES           (determine-communication-kses ?this-structure)
         GOODNESS           *extremely-high*
         RESOLVING-POWER    *zilch*
         RUNTIME            '(0 0)
       ))
     SUBGOALING
       '(((vt) (((vl gl sl)    ((1 10000)))))
         ((pt) (((vt vl gl sl) ((1 10000)))))
        )
   )

 CHARACTERISTICS
   (LOCATION    (rectangle-center region-s)
    NODE-SPEED  '((1 *all))
   )

 RELATIONS
```

This entry refines the communication relation between synthesizing nodes and sensing nodes from the synthesizing node perspective.

```
  ((RELATION-NAME  sensor-data
    RELATION-TYPE  refine
    CONSTRAINT
       (PARTNERS       (sensor)
        RESTRICTIONS
             ((sensor  (and  (compatible-communication?
                                  ?this-relation  ?partner-relation)
```

This constraint (or conjunct of the constraint) determines if a sensing node is capable of detecting signals within the region for which the synthesizing node is responsible.

```
                        (sensor-scans-part-of-rectangle?
                           region-s  ?partner-structure)))
            )
      GROUPS
```

This group constraint returns a list of those groups of sensing nodes that together are able to scan the entire region for which the node is responsible. No redundant sensing nodes are included in a group.

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

This preference constraint selects the smallest group in the list.

```
            ((sensor  (select-smallest-set ?groups))
            )
       )
    ADDITIONAL-DATA
      ((communication
         ((DIRECTION          receive
           NATURE             (hyp)
           DISPATCHES
```

Synthesizing nodes always accept information from sensing nodes.

```
            ((CONDITION          t
                  INFORMATION        (sl-sensor-regions)
                 ))
             ))
         ))
     )
```

This entry refines the communication relation between synthesizing nodes and integrating nodes.

```
     (RELATION-NAME  to-integrator
      RELATION-TYPE  refine
      CONSTRAINT
         (PARTNERS      (integrator)
          RESTRICTIONS
```

This constraint accepts all members of the `integrator` *domain.*

```
                 ((integrator  t)
                 )
          GROUPS
```

Create groups of size one from all of the candidates.

```
                 ((integrator  (any-candidate ?candidates))
                 )
          PREFERENCE
```

Select the integrating node that is nearest.

```
                 ((integrator  (located-nearest ?this-structure ?groups))
                 )
         )
```

```
ADDITIONAL-DATA
  ((communication
      ((DIRECTION          send
        NATURE             (hyp)
        DISPATCHES
          ((CONDITION      (above-threshold 0)
            INFORMATION    (vt-sensor-regions)
          ))
       )
       (DIRECTION          receive
        NATURE             (goal)
        DISPATCHES
          ((CONDITION      (above-threshold 0)
            INFORMATION    (vt-sensor-regions)
          ))
      )
```

The DVMT *provides for messages that are to be sent only in response to certain requests, thus the* reply *direction.*

```
       (DIRECTION          reply
        NATURE             (hyp)
        DISPATCHES
          ((CONDITION      (above-threshold 0)
            INFORMATION    (vt-sensor-regions)
          ))
      ))
    ))
 ))

); end "synthesizing-node".

;;;; ***************************************************************************
```

The last individual structure, vanilla-sensor, is described below. It is instantiated for the sensor-array component in the hierarchical description.

```
;;;; ***************************************************************************

(NAME  vanilla-sensor
```

This individual structure is of the DVMT sensor *type.*

```
 TYPE  sensor

 PARAMETERS
   ((PARAMETER-NAME region-v               DEFAULT '(0 0 22 22))
    (PARAMETER-NAME max-nodes-per-sensor  DEFAULT 2)
   )

 LOCAL-VALUES
```

There are two ways of giving the position of a rectangular region in the DVMT. The first is with a list of four integers that are the minimum x and y coordinates followed by the maximum x and y coordinates; this is "corner" notation and is the most used. The other method is to give the location of a point in the region and then specify the minimum and maximum coordinates of the rectangle in terms of x and y displacements from the point. This "displacement" notation is used in the DVMT *when describing the region scanned by a sensing node. It is sometimes necessary to convert from one notation to the other.*

```
   ((scan-displacements
         (second (convert-coords$corner-to-displace 'region-v)))
    (sensor-region
        '(ABSTRACTION-LEVELS '(sl)
          INTEREST-REGION    'region-v
          TIME-PERIODS       *all
```

The SIGNAL-CLASS-MASK *field in the* CHARACTERISTICS *field of descriptions of sensing nodes (see below) specifies which signal classes the node can detect. These events are listed here as well.*

```
            EVENT-CLASSES      (list-detected-signal-classes ?this-structure)
           ))
  )
```

In the DVMT, *the responsibilities and resources of a sensing node need no explicit description.*

```
RESPONSIBILITIES  nil

RESOURCES         nil

CHARACTERISTICS
   (LOCATION            (rectangle-center 'region-v)
```

In the DVMT, *sensing nodes are treated as black boxes: no attempt is made to simulate how they produce their results. Consequently, their descriptions are devoted to providing "statistical" information used to model their behavior.*

```
    LOCATION-ACCURACY   *extremely-high*
    CLASSIFY-ACCURACY   *extremely-high*
    PERFORMANCE-RATING  *extremely-high*
```

A one in the list indicates that the sensing node is capable of detecting the signal type associated with the one's location (this is DVMT *notation).*

```
    SIGNAL-CLASS-MASK
       (0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0)
```

The value in MULTIPLE *indicates how many signals the sensing node will detect, per stimulus signal, within the region given in the* DISPLACEMENTS *field; thus the closer this number is to one, the more accurate the sensing node. (The* DISPLACEMENT *value is implicitly combined with the value of the* LOCATION *field, above, to make a complete specification of a rectangular region in displacement notation.)*

```
      SCAN-DATA
        ((DISPLACEMENTS  scan-displacements
          MULTIPLE       1
        ))
    )
```

```
 RELATIONS
```

This entry refines the communication relation between synthesizing nodes and sensing nodes that was declared in the `hierarchical` *structure.*

```
   ((RELATION-NAME  to-synthesizer
     RELATION-TYPE  refine
     CONSTRAINT
        (PARTNERS      (node)
         RESTRICTIONS
              ((node  (and (compatible-communication?
                                  ?this-relation  ?partner-relation)
```

This conjunct of the constraint tests for the pathological case in which a potential partner in the relation does not have the resources necessary for processing the information sent to it by the sensing node.

```
                             (node-has-ks-names?
                                  '(*signal-processing)  ?partner-structure)))
              )
         GROUPS
```

This group constraint returns a list of all of the subsets of `?candidates` *with one to* `max-nodes-per-sensor` *members.*

```
              ((node  (subsets-to-size-n max-nodes-per-sensor ?candidates))
              )
          PREFERENCE
              ((node  (select-smallest-set ?groups))
              )
         )
       ADDITIONAL-DATA
         ((communication
            ((DIRECTION            send
              NATURE               (hyp)
              DISPATCHES
                ((CONDITION        t
                  INFORMATION      (sensor-region)
                ))
            ))
         ))
    ))

) ; end "vanilla-sensor".

;;;; ***************************************************************************
```

Appendix B

Domain Specific Functions

This section 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 test the contents of a particular field in a given description or collect or summarize information about the fields in a description. Examples of this class of function have already been seen. They are `node-has-ks-names?` and `sensor-scans-part-of-region?`. `node-has-ks-names?` tests if a node's knowledge sources include a particular set of knowledge sources. `sensor-scans-part-of-region?` determines if a sensor is capable of detecting vehicular activity in some portion of a given region. Other examples of this class of function examine the contents of a sub-field of the CHARACTERISTICS field in a sensor description to see if it includes the given list of values (`sensor-detects-signals?`), compare the overall rating of a sensor's accuracy at classifying and locating signals with a given value (`minimum-sensor-accuracy?`), test to see if an organization has interest-areas that intersect with a supplied list of interest-areas in all dimensions except their regions—which are allowed to lie side by side if they do not overlap—(`bordering-interest-areas?`), list the classes of signals a sensor detects (`list-detected-signal-classes`), and determine the communication knowledge sources an individual will need based on its communication activity (`determine-communication-`

kses).

Another function class tests for the presence of a relation between fields in multiple descriptions or summarizes data from multiple descriptions. The same example in section 2 includes an example of this class of function as well.`sensors-that-cover-region` finds all of the combinations of sensors (from a list of sensors) that will, between them, scan all of a given region.`within-distance?`, `nearest-n`, and `sum-sensors-scan-regions` are three other functions from this class. The first checks that the distance between two locations does not exceed a given value; the second orders individual organizations by their distance from a given location and returns a list of the *n* closest organizations; the last returns a region that encloses all of the regions scanned by a list of sensors.

One class of functions are those that are essentially operations on sets. Among these are `subsets-to-size-n`, which returns all of the sub-sets of a set that contain *n* or fewer members, and `select-smallest-set` which selects from a set of sets the one with the fewest members (or one of them if there are several of a size equal to the smallest).

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 are `minimum-enclosing-rectangle`: takes a list of rectangles and returns the one that surrounds them all (used in `sum-sensors-scan-regions`); `intersecting-rectangles?`: takes a list of rectangles and returns the rectangle that is overlapped by all of them; `list-uncovered-sub-regions`: takes two overlapping rectangles, breaks up the area of the first rectangle that is not overlapped by the second into smaller rectangles (at most, three are required), and returns them in a list (used by `sensors-that-cover-region`); `fill-rectangle-with-overlapping-rectangles`: takes a rectangle, the dimensions of another rectangle that is to be repeatedly overlaid upon the first, the number of overlaying rectangles that are to be used, and the amount of overlap as parameters, then returns the list of overlying, overlapping, rectangles (since an attempt to specify all of these parameters may result in a situation that cannot by physically realized, one parameter may be left unspecified, the function will then compute the missing value). This last function is used by a composite structure to assign regions to its components.

The final function class is for unusual operations on integers. It has just one example, `group-divide`. Its arguments are an integer, a maximum divisor, and a minimum quotient. It 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 sub-hierarchies (in the example of section 3) where: no manager wants to manage more than some number of sub-hierarchies (maximum divisor), and a minimum number of employees are required to make up a hierarchy (minimum quotient).